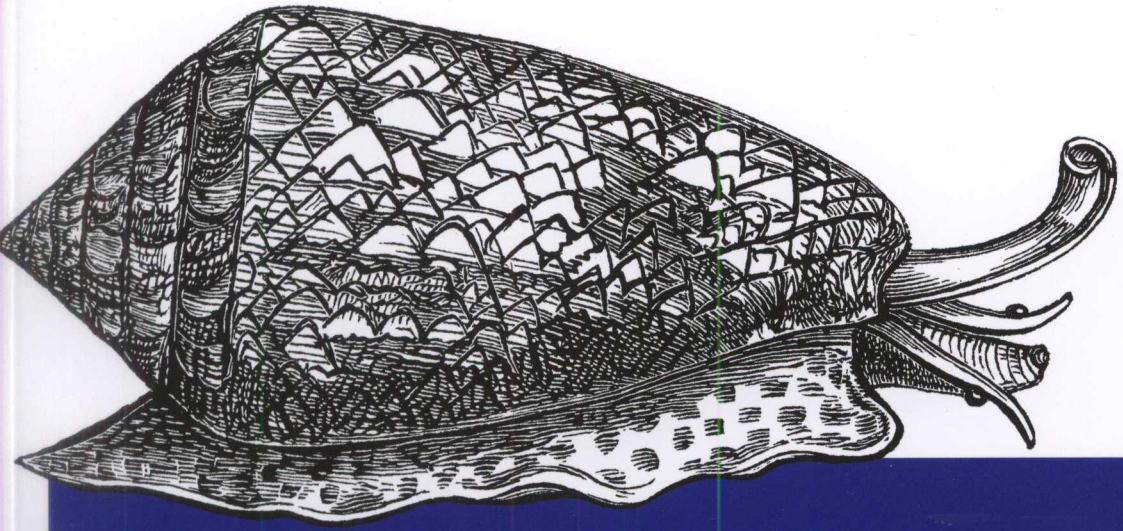
O'REILLY®

Broadview®
www.broadview.com.cn



可伸缩架构

面向增长应用的高可用

Architecting for Scale High Availability for Your Growing Applications

[美] Lee Atchison 著
张若飞 张现双 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

可伸缩架构
面向增长应用的高可用

Architecting for Scale
High Availability for Your Growing Applications

[美] Lee Atchison 著
张若飞 张现双 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

随着互联网的发展越来越成熟，流量和数据量飞速增长，许多公司的关键应用程序都面临着伸缩性的问题，系统变得越来越复杂和脆弱，从而导致风险上升、可用性降低。本书是一本实践指南，让IT、DevOps和系统稳定性管理员能够了解到，如何避免应用程序在发展过程中变得缓慢、数据不一致或者彻底不可用等问题。规模增长并不只意味着处理更多的用户，还包括管理更多的风险和保证系统的可用性。作者Lee Atchison 在可用性、风险管理、服务和微服务、扩展应用程序和云服务方面提出了一些技巧，使得我们在构建各类应用程序时，既能够保证产品的质量，又能够处理海量的流量、数据以及需求。

如果你管理着软件开发人员、系统可靠性工程师、DevOps工程师，或者你经营着一个拥有大规模应用程序和系统的机构，本书中所提供的建议和指导都能够帮助你，让你的系统运行得更加平稳和可靠。

版权贸易合同登记号　图字：01-2016-6771

图书在版编目（CIP）数据

可伸缩架构：面向增长应用的高可用/（美）李·艾奇逊（Lee Atchison）著；张若飞，张现双译. —北京：电子工业出版社，2017.7
书名原文：Architecting for Scale：High Availability for Your Growing Applications
ISBN 978-7-121-31684-5

Ⅰ. ①可… Ⅱ. ①李… ②张… ③张… Ⅲ. ①数据管理—研究 Ⅳ. ①TP274

中国版本图书馆CIP数据核字（2017）第120534号

策划编辑：张春雨
责任编辑：刘　舫
封面设计：Karen Montgomery　张　健
印　　刷：三河市良远印务有限公司
装　　订：三河市良远印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路173信箱　　邮编：100036
开　　本：787×980　1/16　　印张：12　　字数：263千字
版　　次：2017年7月第1版
印　　次：2017年7月第1次印刷
定　　价：65.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。
质量投诉请发邮件至zlts@phei.com.cn，盗版侵权举报请发邮件至dbqq@phei.com.cn。
本书咨询联系方式：010-51260888-819　faq@phei.com.cn。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——Wired

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去 Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

互联网从一穷二白发展到今天，“可伸缩性”、“水平扩展”这些以前只有大型互联网公司才面临的技术挑战，现在任何一家互联网公司都需要想办法去解决。在我看来，写好一段代码不难，写好一个单机软件也不难，但是要想写好一个伸缩性良好的大规模分布式系统却很难。这其中最难的地方首先在于思想观念的转变。程序员一开始接触到的就是一个确定的概念和世界，当程序逻辑一定时，什么样的输入就应该有什么样的输出。但是在面对大规模系统时，我们首先要承认的是不确定性，即有很多我们想不到、料不到的事情会发生，一个边角的小错误可能就会导致整个系统的全面雪崩。因此，我们不能再把系统当作一个稳定的、确定的程序看待，而需要在设计时充分考虑到那些可能不确定的情况。其次，就是管理整个系统的有效手段。随着规模的不断增长，之前的人工方式已经明显无法跟上机器的增长速度，规模增长所带来的可靠性、可用性问题时刻都在挑战着整个团队的极限。因此，如何有效地评估、预测、管理、监控一个大规模的系统，这件事本身已经变成一个复杂的系统性工程，需要有效的方法论、原则、工具以及最佳实践经验的相应支撑。

当一个系统规模开始扩张时，可用性往往是系统首先要面临的问题。在激增的流量面前，每一次系统处理缓慢、崩溃都会给公司的业务带来损失，因此理解什么是高可用性对系统来说至关重要。为了维持或者提高系统的可用性，我们需要一套标准和手段，来测量系统的可用性，从而才能在第一时间发现系统可用性是否发生了变化，并制订相应的解决方案。

随着系统规模不断扩大，各种不确定的因素会被放大，这些不确定性会给系统带来更多的不确定性。在构建大规模可伸缩系统的时候，需要先梳理清楚系统中存在哪些不确定性，把它们定义成风险因素，并建立相应的风险模型，定期回顾并确保更新新的风险点。在这之后，我们要针对每个风险点制订相应的应对方案和措施，并持续对风险管理计划、缓和计划和容灾计划进行持续测试和评估，这样才能避免在风险发生时手忙脚乱。

当系统规模发展到一定程度时，系统本身的架构就会突显瓶颈，之前的单体系统已经难

以再开发和维护。这时候，不管从技术角度出发，还是从团队人员角度出发，服务化都是一个必然的趋势。我们需要将原有的系统功能和逻辑，拆分成多个独立的服务来进行管理。但是，随着服务越来越多，如何管理各个服务、服务间如何通信，以及团队的划分都会带来新的问题。随着业界在这方面的经验越来越多，经过 Netflix、Amazon 等公司的实践和推广，微服务的概念也随之出现，并提供了完善的解决方案和工具。

在解决这些问题之后，我们会发现，系统规模再继续发展就会遇到 IDC 的限制，单个数据中心甚至没有空间来存放服务器，也支持不了所需的网络带宽。幸运的是，云计算和服务的出现为我们创造了无限的可能。我们无须再考虑硬件和场地的问题，无须再处理烦琐的运维事项，可以按照使用量来付费，甚至很多服务也无须自己搭建，可以直接使用云服务厂商提供的各种高性能、高可用服务。云计算给我们带来的是一种颠覆性的变化，不光影响到我们的开发、运维、技术栈，更深远地影响到我们对技术和架构的思考方式，进而为创造更多商业价值提供可能。

本书作者 Lee Atchison 是 New Relic 的首席云架构师和布道师，负责领导搭建公司的基础设施产品，并且帮助公司设计了一个稳定的、基于服务的系统架构。他曾在 Amazon 担任了 7 年高级技术经理，深刻了解如何设计基于云的、可伸缩的系统，主导创建了 AWS Elastic Beanstalk 等产品。在本书中，Lee 根据自身多年的经验，为我们分享了大规模可伸缩系统设计和实现中应该注意的几个方面，这其中的方法和技巧都是经过时间考验的无价经验。相信各位读者看完此书后，不再惧怕那些未知的困难，能够在保持系统规模增长的同时，保证系统的高可用性和可伸缩性。

愿以后的每个系统都是高可用、可伸缩的系统。

目录

第 II 部分　风险管理

第Ⅵ部分 总结

序

我们生活在一个有趣的时代，可以称它是一个软件的寒武纪大爆炸。在这个过程中，构建新系统的成本呈数量级式下降，同时系统之间的关联程度也呈同等数量级的增长。借助于 Amazon 的 AWS、微软的 Azure 和 Google 的 GCP 等资源，我们可以将系统在物理上扩展到一个几年前还只能想象的规模。

这些资源及其似乎无限的能力，正在以各种前所未见的方式，将新的思想、产品和市场极其快速地传播出去。但是，只有当我们构建的系统可以保持扩展的同时，所有这些探索才能成为可能。与以前相比，虽然构建小型系统变得容易很多，但是构建一个可以快速、可靠扩展的系统，并不像增加更多的硬件和存储空间那么容易，实际证明，这要难得多。

每个软件系统都会经历一个可预见的生命周期，从一个人能够完全理解的、小型的、设计精妙的解决方案，迅速增长为一个积累了大量技术债务的庞大系统，随后又逐渐分裂成由一些不完善的服务随机组成的组合，并最终演化成在广度（更多用户）和深度（更多功能）方面均可稳定扩展的、设计良好的分布式系统。对于这样的系统来说，我们很容易从外部了解要做哪些事情（让它变得更加可靠！），但又很难了解它内部的细节。幸运的是，本书是一本关于这方面不可或缺的指南，从可用性到服务层，从比赛日到风险模型，Lee 一步步介绍了影响大规模系统的各个关键因素和实践方式。

Lee 加入我们的时候，是 New Relic 第一次从仅拥有一个产品正在向多个产品转型的时期，当时我们正沉浸在用户极速增长和公司成功的喜悦中。Lee 的到来，为我们带来了他在 Amazon 的丰富经验，不管是零售业务还是 AWS 业务都曾经历过巨大的增长。Lee 曾是这些团队的领头人，曾经积极参与过与可扩展性有关的所有事情，也遇到过很多失败。对我们来说，幸运的是，他已经经历过这些挫折与困苦，其中的教训可以让我们避免再犯同样的错误。

在 Lee 加入 New Relic 之前，多年以来，我们一直经历着系统服务不可用的尴尬处境。我们原有的庞大系统也逐渐无法支持业务的发展，不管是可用性、可靠性还是性能都不

是很好。但是，通过充分运用 Lee 在本书中所写的各项技巧，我们逐渐克服了这些困难，并构建了如今稳定可靠的企业级服务。其中我们使用的一个工具，建立了可用性工程的四个级别：青铜、白银、黄金和白金。要达到青铜级，团队必须拥有风险模型以及预定义的 SLA 标准。要达到白银级，团队必须能够监控风险模型中标识出来的问题，并使用比赛日的方式来解决。黄金级意味着风险已经被缓解掉了。白金级如同 CMM 5 级一样，不仅系统可以自愈，而且我们关注持续性的改进。我们首先集中精力对第一级的服务进行改进，然后上升到第二级的服务，逐步推进，最终使得所有团队都至少达到了白银级，并且大多数团队通过了黄金级，甚至有几个团队达到了白金级。

后来我加入了 InVision App 这个更年轻的公司。我又一次经历着从早期成功向高速增长的过程，一直推动大家去使用 Lee 之前带给我的技术和工具。在这个新系统、新产品、新公司的爆炸年代，我强烈建议大家跟我做一样的事：向 Lee 学习如何构建可伸缩的系统。

——*Bjorn Freeman-Benson* 博士，
InVision App 首席技术官

前言

当应用程序开始增长时，通常会出现两件事情：它们明显变得更加复杂（也更加脆弱），并且需要处理显著增加的流量（需要更先进、更复杂的管理机制）。这会让应用程序逐渐陷入一个死亡漩涡，用户会不断经历着限流、宕机以及其他服务质量和可用性方面的问题。

但是你的用户不会关心这些。他们只希望使用应用来做他们希望做的事情。如果你的应用程序宕机、响应缓慢或者信息不一致，用户会马上抛弃你，转而投靠能够帮助他们处理生意的竞争对手。

本书可教会你一些如何构建并管理大规模应用程序的基本技巧，帮助你避免陷入如上所述的死亡漩涡。一旦你掌握了这些技巧，你的系统就能够可靠处理海量的流量，从容面对流量中大量的不确定性，同时不会对用户期望造成任何影响。

本书的读者对象

本书的目标读者包括构建和管理大规模应用程序和系统的软件工程师、架构师、技术经理以及总监。如果你管理着软件开发人员、系统可靠性工程师、DevOps 工程师，或者你经营着一个拥有大规模应用程序和系统的机构，本书中所提供的建议和指导都能够帮助你，让你的系统运行得更加平稳和可靠。

如果你的应用程序已经从很小的规模变得很大（并且正在经历着增长所带来的各种问题），你可能正在为系统的低可靠性和低可用性烦恼。如果你正在头疼如何管理技术债务以及相关的系统故障，本书恰好提供了这些方面的指导，能够帮助你通过降低技术债务，让应用程序更轻松地扩展到更大规模。

编写本书的原因

在 Amazon 零售和 AWS 业务多年从事构建高可伸缩应用程序之后，我加入了 New Relic 这个正在迅速成长的公司。当时，New Relic 已经感受到了因为缺少管理高可伸缩应用程序的系统、流程所带来的痛苦，但是尚未完整形成能够扩大其应用程序规模的流程和规范。

在 New Relic，我亲眼目睹了一个公司在扩张规模过程中所经历的痛苦与挣扎，同时也意识到，还有很多其他公司每天都在经历着这些痛苦。

编写本书的初衷，是为了帮助那些正在面对其应用程序高速增长的人们，使其了解到一些有用的流程和最佳实践，避免他们再掉入规模扩张过程的各种陷阱之中。

无论你的应用程序每年增长十倍还是百分之十，也无论增长的是用户数量、交易数量、数据存储量还是代码复杂性，本书都可以在构建和维护应用程序方面为你提供帮助，让它们在保持高可用性的前提下实现增长。

现在我们所说的规模

基于云的服务正在以极快的速度增长和扩张。这主要归功于对云服务的大量需求，“软件即服务(SaaS)”逐渐成为应用程序开发的标准。由于 SaaS 应用程序天生多租户的特点，它们对于规模上的问题尤其敏感。

随着世界的改变，以及我们对 SaaS 服务、云服务、海量应用程序越来越多的关注，规模性问题也变得越来越重要。似乎我们看不到，云应用程序在体积和复杂性方面会出现增长到头的情况。

关键的机制在于，我们当前用来管理大规模系统的前沿科技，很可能会成为明天的基础工具，而明天我们可能遇到的规模问题，也会让当前的解决方案相形见绌。我们需要越来越复杂的系统和架构，来处理将来可能无限增长的规模。

本书的目的，就是为了提供可以禁得起时间考验的知识。

本书导读

管理规模并不只是管理流量，还包括管理风险和可用性。通常来说，所有这些东西都是描述相同问题的不同方式，并且它们息息相关。因此，为了能够合理地讨论规模问题，我们还必须考虑到可用性、风险管理以及像微服务和云服务这样的现代架构模型。

因此，本书按照如下章节来划分内容。

第 I 部分：可用性

当某个应用程序开始扩张规模时，可用性和可用性管理通常是最先受到影响的部分。

第 1 章，什么是可用性

为了更好地让读者理解，我们会讲解一下高可用性的意义，以及它与可靠性之间的区别。

第 2 章，提高应用程序可用性的五个要点

在本章中，我针对如何提高应用程序的可用性提出了五个核心方向。

第 3 章，测量可用性

本章会介绍一种测量可用性的标准算法，并进一步讲解高可用性的作用和价值。

第 4 章，提高下降的可用性

如果你的应用程序正遇到可用性的问题（或者你想知道这是否正在发生），我们提供了一些管理手段，来帮助你提高应用程序的可用性。

第 II 部分：风险管理

理解系统中的风险，对于提高应用程序的可用性，以及它后续向更大规模发展的能力是至关重要的。

第 5 章，什么是风险管理

本章会通过介绍风险管理的基本知识，引出如何管理高可伸缩应用程序的话题。

第 6 章，可能性与严重性

本章会讨论风险发生时的严重性与可能性之间的区别。它们都非常重要，但是方式不同。

第 7 章，风险模型

在本章中，我会以一个精心设计的系统为例，来帮助你理解和管理系统中的风险。

第 8 章，风险缓和

本章会讨论如何处理系统中已知的风险，并减少它们对应用程序的影响。

第 9 章，比赛日

本章会介绍如何对风险管理计划、风险缓和计划以及容灾计划进行持续的测试和评估。本章回顾了在生产环境实现比赛日所需的技术，以及它所带来的好处。

第 10 章，构建低风险系统

在本章中，我会给出如何降低应用程序中的风险，以及如何构建低风险系统的建议。

第III部分：服务和微服务

服务和微服务都是一种架构方案，用于构建需要更大规模运行的、更加大型且复杂的应用程序。

第 11 章，为什么使用服务

本章会介绍为什么服务对于构建高度可扩展的应用程序如此重要。

第 12 章，使用微服务

在本章中，我会介绍如何创建基于微服务的架构，主要关注于如何对服务大小进行合理分割，确定服务的边界，以便提高可扩展性和可用性。

第 13 章，处理服务故障

在本部分的最后一章中，我们会来讨论如何构建能够处理故障的服务。

第IV部分：如何让应用程序具有伸缩性

“伸缩”其实不仅仅与流量有关，它关系你的组织，以及你的组织如何来响应更大的应用程序需求。

第 14 章，两次失误的高度

本章会介绍如何在出现故障的情况下，依然能够通过伸缩系统来保持高可用性。

第 15 章，服务所有权

本章会讲解，关注服务的所有权，能如何帮助你扩展组织和应用程序。

第 16 章，服务分级

本章会介绍如何区分各个服务的关键程度，从而帮助管理对服务的期望。

第 17 章，使用服务分级

当制订服务分级制度之后，我们会介绍如何通过它来管理服务故障的影响、响应性需求以及期望管理。

第 18 章，服务等级协议

在本章中，我们会讨论如何使用 SLA 来管理服务所有者之间的相互依赖。

第 19 章，持续改进

本章会就如何改进应用程序整体的可扩展性，提供相应的技术和指导。

第 V 部分：云服务

对于构建和管理需要极强可伸缩能力的大型、关键性系统来说，基于云的服务正在变得日益重要。

第 20 章，变化和云服务

本章介绍了云服务对构建高度可伸缩的 Web 应用程序所带来的改变。

第 21 章，云上的分布

本章概述了如何有效使用地域和可用区来提高可用性和可伸缩性。

第 22 章，托管的基础设施

本章会介绍如何使用托管服务（例如 RDS、SQS、SNS 和 SES）来伸缩应用程序，同时降低管理方面的工作。

第 23 章，云资源分配

本章中我们会讨论如何分配云资源，以及不同分配技术对应用程序可伸缩性的影响。

第 24 章，可伸缩的计算选项

本章会介绍一些高度可伸缩的编程模型，例如 AWS Lambda。你可以通过这些技术来提高可伸缩性、可用性以及应用程序的可管理性。

第 25 章，AWS Lambda

本部分的最后一章会更加深入地介绍 AWS Lambda 技术，它可以为简单的计算型需求提供高度的可伸缩性。

第 VI 部分：总结

第 26 章，融会贯通

本章会将之前每个部分的核心要点汇总成一个简短的总结，帮助你来回忆每章的内容。

在线资源

本书的网站（*http://www.architectingforscale.com/*）提供了一些额外的信息，包括补充材料的链接地址。你可以在这个网站上找到更多关于我的信息，也可以关注我的博客。

本书使用的约定

本书使用以下一些排版约定：

斜体（*Italic*）

表示新的名词、URL、电子邮件地址、文件名以及文件扩展名。

等宽字体（`Constant width`）

用于程序清单，以及段落中对变量、函数名、数据库、数据类型、环境变量、语句和关键字等程序元素的引用。

等宽加粗字体（**`Constant width bold`**）

用于显示命令或者其他需要由用户输入的文字。

等宽斜体字体（*`Constant width italic`*）

用于显示需要由用户提供的值或者由上下文决定的值所替换的文本。

该图标表示一个提示或者建议。

该图标表示一个一般说明。

该图标表示一个提醒或者警告。

中文版书中切口以“<⬭>”表示原书页码，便于读者与英文原版图书对照阅读，本书的索引中所列的页码也为英文原版图书中的页码。

Safari Books Online

Safari Books Online（*www.safaribooksonline.com*）是一个按需出版的数字图书馆，出版各种专业的书籍和视频，它们由世界上技术和商业领域的优秀作者撰写。

技术人员、软件开发者、网页设计者及商业和创意专业人士都把 Safari Books Online 当作科研、问题解决、学习及认证训练的主要资源。

Safari Books Online 为组织、政府机构和个人提供了大量的产品组合和定价方案。

订阅者可以从一个完全可搜索的数据库中获取成千上万的书籍、培训视频和尚未出版的手稿，这些出版商包括 O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology，以及其他数十家出版社。如想了解更多 Safari Books Online 的信息，请在线访问我们。

如何联系我们

请将对本书的评价和存在的问题通过如下地址告知出版者：

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网站，你可以在那里找到关于本书的相关信息，包括勘误列表、示例代码以及其他信息。本书的网站地址是：

http:/bit.ly/architecting-for-scale

对于本书的评论和技术性的问题，请发送电子邮件到：

bookquestions@oreilly.com

关于我们的书籍、课程、会议和新闻的更多信息，请参阅我们的网站 *http://www.oreilly.com*。

在 Facebook 上找到我们：*http://facebook.com/oreilly*
在 Twitter 上关注我们：*http://twitter.com/oreillymedia*
在 YouTube 上观看我们：*http://www.youtube.com/oreillymedia*

致谢

虽然我无法一一列举出所有为本书出版做出贡献的人，但是我还是想特别感谢以下帮助过我的人。

- Bjorn Freeman-Benson，在开始编写本书时，他给了我很多支持，并且提供了加入 New Relic 的工作机会，让我能够去思考如何编写本书。
- Kevin McGuire，我们既是朋友也是知己。我们从 New Relic 开始就在一起工作，他的远见和想象力为我的职业生涯指明了方向，并一直指导我到今天。
- Natasha Litt，她是我的好朋友，同时也给了我很多鼓励和支持。
- Jade Rubick，他始终保持着一贯的笑容和积极的态度，为我提供了很好的建议和指导。有幸与你为友。
- Jim Gochee，是他向我介绍了 New Relic 这个产品，最终也成为我的工作。
- Lew Cirne，是他的远见带给了我们 New Relic，也带给了我一份工作和一个家庭。每次在跟 Lew 进行一对一的谈话后，我都会被他的幽默和热情所感染和打动。怪不得 New Relic 会如此成功。
- Abner Germanow、Jay Fry、Bharath Gowda 和 Robson Grieve，他们为我争取到了在 New Relic 工作的机会。谁说我就一定不能胜任这个职位呢？相反，我做得还非常好。这是目前为止我感到最开心、最有收获，以及对我个人来说最充实的一份工作。
- Mikey Butler、Nic Benders、Matthew Flaming，以及其他 New Relic 的技术主管，感谢他们多年以来的支持。
- Kurt Kufeld，他曾是我的导师，感谢他帮助我融入 Amazon 怪异、混乱的工作环境，让我不断接受挑战，拼尽每一分努力，并最终获得了巨大的成功。
- Greg Hart、Scott Green、Patrick Franklin、Suresh Kumar、Colin Bodell 和 Andy Jassy，他们给了我在 Amazon 和 AWS 工作的机会，之前从未想过。
- Brian Anderson，我的编辑，他给了我编写本书的机会，并且在写书的每个阶段都在帮助我。

我想要特别感谢 Abner Germanow 和 Bjorn Freeman-Benson，是他们让我有机会能够编写本书，没有他们，也不可能有本书的存在。出于这个原因，我会永远感谢他们。

感谢我的家庭，尤其是我的妻子 Beth，她永远是指引我生活方向的灯塔。与她在一起，我的生活变得越来越光明，我前进的道路也越来越清晰。

对于所有帮助过我的人们，以及所有我没有提到的人们，我衷心地感谢你们。

最后，我还必须介绍一下那些毛茸茸的小家伙们：Izzy，爱打鼾的西班牙猎犬；Abby，快乐的柯基犬；以及 Budha，一只好动的小猫咪，它给我带来的可不仅仅是本书中的错别字而已。

第 I 部分

可用性

没有高可用性，就没有可伸缩性。

第1章

什么是可用性

如果你的系统有很多不错的功能但是没法使用，那么没人会关心你的系统本身如何。

在设计面向可伸缩的系统架构时，最重要的一个话题就是可用性。虽然有些公司和服务认为一定的宕机时间是合理且允许的，但是对于大多数业务来说，宕机不可能不影响到用户的满意度，甚至最终会影响到公司的发展。

以下是所有公司在确定系统可用性对公司和用户的重要程度时，必须要问自己的一些基本问题。这些问题本身，以及其显而易见的答案，就是为什么可用性对可伸缩性如此重要的核心：

- 如果用户在需要使用你的服务的时候却不能用，他们为什么要买你的服务？
- 如果用户在需要使用你的服务的时候却不能用，他们会有什么样的想法和感受？
- 如果你的服务不可用，你如何能让用户感到高兴，给公司赚钱，达成你的商业承诺和目标呢？

只有系统是可用的，才有可能让用户感到高兴并愿意使用你的系统。因此，系统可用性和用户满意度之间存在着一个直接、重要的联系。

高可用性对于构建高可伸缩系统是一个极其重要的因素，因此我们在本书中会花费很长时间来讨论这个话题。当面临大量的业务需求时，你如何构建一个高可用的系统（服务、应用程序或者环境）呢？

在本章中，我们会定义什么是可用性，以及它与可靠性之间的区别。我们会在后续章节中来讨论，可用性在构建高可伸缩系统中所扮演的角色。

重大赛事

今天是周日——观看重大赛事的日子。你邀请了最好的20位朋友到家里来，打算在新买的300英寸Ultra Max TV上观看这次比赛。所有人都来了，你的房子里放满了零食和啤酒，每个人都在大笑。比赛就要开始了，然后……

灯灭了……

电视不亮了……

对于你和你的朋友来说，比赛也结束了。

失望至极。你拨打当地电力公司的电话。服务人员冷冷地说："很抱歉，我们只能保证当地电网95%的可用性"。

为什么可用性很重要？因为你的用户期待你能在任何时间都提供服务。任何小于100%可用性的系统都会对你的业务带来灾难性的后果。

可用性与可靠性

可用性与可靠性是两个很相似但完全不同的概念。理解这两者之间的区别，对于理解本书后续内容至关重要。

在我们的语境中，可靠性一般指一个系统的质量。通常来说，它意味着一个系统能够按照技术标准持续运行的能力。如果一个软件通过了所有的测试套件，并且基本完成了它应该做到的事情，那么我们可以说它是可靠的。

而我们所提到的可用性，一般指系统在其能力范围内执行任务的能力。系统是否运转正常？是否可以操作？是否可以响应？如果答案是"是"，那么系统就是可用的。

如你所见，可用性和可靠性非常相似。如果系统不可靠，它也很难是可用的，如果它不可用，也很难是可靠的。

但是，一般来说，当我们谈论可靠性和软件时，我们指的是软件执行应有功能的能力。大体上来说，可靠性的主要指标是软件是否通过了所有的测试套件。

除此之外，当考虑可用性的时候，我们实际考虑的是系统是否正在"运行"以及是否能够提供功能服务。如果我们向它发起一个查询请求，它是否能进行响应。

5 以下是我们对这些术语的定义。

可靠性

系统是否具备无差错地执行预期操作的能力。

可用性

为了执行这些操作，系统当前可运行的能力。

一个计算 2+3 却得到结果 6 的系统，其可靠性很低。一个计算 2+3 却永远不返回结果的系统，其可用性很低。可靠性通常可以通过测试来修复，而可用性通常较难解决。

你会因为在系统中引入了一个 bug，导致计算 2+3 得到了 6。这通过测试套件可以很容易发现并修复。

但是，假设你的应用程序可以很稳定地计算 2+3=5。现在假设它运行在一个网络连接不好的计算机上，结果如何？有时它可能返回 5，有时它可能什么都不返回。这时候，应用程序是可靠的，但它不是可用的。

在本书中，我们主要关注于如何设计高可用的系统架构。我们会假设你的系统是可靠的，假设你知道如何构建和运行测试套件，我们只会在可靠性对系统架构或者可用性造成直接影响时才讨论它。

什么导致了低可用性

究竟是什么原因导致之前还运行正常的系统，却慢慢变得不可用了呢？通常，这会有许多种原因。

资源耗尽

用户数量的增加，会导致系统使用的数据量增加，从而可能导致系统资源耗尽，应用程序运行越来越慢并最终无法响应。

预期之外的压力变化

随着应用程序被越来越多的人使用，可能需要对代码和应用程序进行修改以支撑不断增加的压力。这些改动通常都是在最后一刻被草草实现，缺乏足够的考虑或计划，因此也增加了问题出现的可能性。

6 流动行为的增加

当应用程序高速发展时，通常需要更多的开发人员、设计师、测试人员和其他人来开发并维护它。大量的个体共同协作会产生大量的流动行为，包括新功能、变更功能，

或者只是一般的维护工作。开发和维护应用程序的人员越多，就会产生越多的流动行为，从而增加了相互之间产生负面作用的可能。

外部依赖

应用程序依赖的外部资源越多，由于这些资源导致的可用性问题就会越多。

技术债务

随着应用程序复杂性的增加，通常会导致技术债务（例如，通常随着应用程序逐渐发展和成熟，对软件未实现的修改和未修复的 bug 数量也会逐渐积累）。技术债务会增加可用性问题出现的可能。

所有快速发展的应用程序都有一个、多个或者所有这些问题。因此，之前运行正常的应用程序，也可能会逐渐出现可用性的问题。通常这些问题会慢慢浮现，有时也会突然发生。

但是发展最快的应用程序都有相同的问题，它们最后都会面临可用性的问题。

可用性问题会增加你的成本，增加你的用户的成本，降低用户对你的信任度和忠诚度。如果系统一直存在可用性问题，你的公司不可能长期存活。

构建可伸缩的应用程序，意味着构建高可用的系统。

7

第2章

提高应用程序可用性的五个要点

构建一个高可用、可伸缩的应用程序不是一件容易的事，也不会是天上掉下来的馅饼。问题总会以你从未预期的方式出现，让你精心设计的功能对所有用户都停止工作。

这些可用性问题通常会在你最想不到的地方出现，甚至一些最严重的问题会来自于最不可能出现的地方。

一次简单的图标故障

这发生在我亲身经历的一个应用程序中，是一次因为忽视依赖故障的典型案例。该程序向用户提供了一个服务，为每个页面顶部提供一个自定义的图标，来表示当前登录的用户。这个图标由一个第三方系统负责生成。

有一天，这个生成图标的第三方系统发生了故障。我们的应用程序却假设该系统总是会正常运行，因此并不知道如何处理这种情况。结果是，我们的应用程序也跟着发生故障。我们整个系统仅仅是因为图标生成这样一个非常小的“功能”，导致无法提供任何服务。

如何才能避免这样的问题呢？如果我们能够预料到第三方系统可能发生故障，就可以在设计过程中考虑到这个故障发生的场景，从而发现我们的应用程序也会随之发生故障。这样，我们就能添加一些逻辑来检查第三方服务，在问题发生时删除图标，或者在问题发生时捕获错误，避免它传递下去并影响页面的其他部分。

一次小小的检查和一些错误恢复机制，就可以帮助应用程序保持正常运行。否则，我们的应用程序就会经历严重的服务中断。

8 所有这一切都是因为缺少了一个图标。

没人有能够预料到问题会在何处发生，也不可能依靠测试来发现所有这些问题。许多问题都是系统性的问题，而不仅仅是代码的问题。

为了发现这些可用性的问题，我们需要后退一步，系统地去了解应用程序的运行机制。以下是五个你可以关注、并且应当关注的要点，它们能够帮助你的系统在规模增长的同时保证高可用性：

- 时刻考虑应对故障
- 时刻考虑如何伸缩
- 缓和风险
- 监控可用性
- 以可预期及明确的方式来处理可用性问题

让我们来详细讲解其中的每一个要点。

要点 1：时刻考虑应对故障

正如 Amazon 的 CTO Werner Vogels 所说，“所有事情每时每刻都会失败”。你应当提前为应用程序和服务发生故障而做出计划。问题迟早会产生。不过现在，我们要讲的是如何解决它。

假设你的应用程序发生了故障，那么它是如何发生的？当你构建系统的时候，应当在设计和实现的方方面面都考虑可用性。例如：

设计

你有考虑过任何设计模式吗？你有使用它们来帮助你提升软件的可用性吗？

通过使用一些设计模式，例如捕获底层异常、重试逻辑和断路器，可以帮助你捕获错误并尽可能避免影响其他功能。这样，你就能够限制问题的影响范围，即使应用程序的某些部分出现问题，依然能够提供其他一些有用的功能。

依赖

如果你依赖的组件出现了故障，你会怎样做？你如何进行重试？如果问题是一个无法恢复的（硬件）故障，你会怎样做？如果是一个可恢复的（软件）故障呢？

断路器模式在处理依赖故障时非常有用，因为它们可以降低依赖故障对你的系统的影响。

如果没有断路器，你可能会因为依赖故障而降低系统的性能（例如，需要一个很长的超时机制来检测故障）。而使用了断路器，你可以“放弃”并停止使用某个依赖，直到你确认它已经恢复了正常工作。

用户

如果出现问题的原因是系统的某个用户，你会怎样做？你能够处理海量的请求吗？你能够限制海量的流量吗？你能够处理传入的垃圾数据吗？如果数据量非常大，你会怎样做？

有些时候，拒绝式服务可能来自于“友方”。例如，用户可能会因为看到一个临时活动，而导致大量请求增加。或者，用户程序中的一个 bug，可能导致他们向你的应用程序拼命地发送请求。如果这样的事情发生了，你会怎样做？流量突增会让你的应用程序宕机吗？或者你能否检测出这种问题，通过限制请求的速度来降低或者消除它们的影响？

要点 2：时刻考虑如何伸缩

你的系统现在正常运行，并不意味着它明天还能够继续正常运行。大多数 Web 应用程序的流量都是在不断增加的。一个今天产生一定流量的网站，明天可能会产生远比你想象大得多的流量。当你构建系统时，不要只考虑当前的流量，要考虑未来的流量。

具体一点，这可能意味着：

- 设计出能够增加数据库数量和容量的架构。
- 考虑限制你的数据伸缩的原因。当数据库达到容量极限的时候会发生什么？你需要确认这些限制因素并在到达极限之前解决它们。
- 你应当能够很容易地添加额外的应用程序服务器。这通常需要仔细考虑在何处和如何来维护状态，以及流量是如何路由的。[注1]
- 将静态流量导向离线提供方。这样你的系统只需要处理必要的动态流量。使用外部的内容分发网络（CDN）不仅可以降低网络需要处理的流量，也能够利用 CDN 的伸缩效率将静态内容更快地分发给用户。
- 考虑是否可以静态生成一些动态资源。通常来说，看上去动态显示的内容实际上大多数是静态的，并且生成静态内容可以让你的应用程序提高可伸缩性。这种“应该静态的动态资源”有些时候隐藏在你想象不到的地方，如下文中所述。

究竟内容应该是静态的还是动态的？

通常，看上去是动态的内容实际上大多数是静态的。设想一个网站上常见的顶部导航栏，绝大多数时候，其中的内容都是静态的，但是偶尔也会出现一些动态的内容。

注 1　这个话题太大了，足以单独写一个章节甚至一本书。

例如，如果你没有登录，页面的顶部可能会显示“请登录”，如果你已经登录了则显示“你好，Lee”（当然前提是你的名称是 Lee）。

这是否意味着整个页面都必须动态生成呢？显然不是。除了页面的登录 / 问候部分，其他部分都是静态的，通过 CDN 可以轻松地进行分发并节省你的计算资源。

当导航栏中大多数内容都是静态内容时，你可以在用户的浏览器中动态地将变更内容添加到页面上（例如根据具体情况添加“请登录”或者“你好，Lee”的内容）。通过将这些动态数据进行分组，并与静态内容加以区分，可以提高 Web 页面的性能，降低应用程序需要处理的动态数据量。这样可以提高可伸缩性，并最终提高可用性。

要点 3：缓和风险

保持系统高可用需要消除系统中的风险。当系统发生故障时，通常我们已经在这之前将故障原因确定为了风险。因此，确定风险是提高可用性的一个重要方法。所有的系统中都存在以下这些风险：

- 存在系统崩溃的风险
- 存在数据库崩溃的风险
- 存在返回结果不正确的风险
- 存在网络连接失败的风险
- 存在新部署的软件功能出现故障的风险

保持系统高可用需要消除风险。但是当系统变得越来越复杂时，消除所有风险也变得越来越不可能实现。保持一个大型系统高可用，更多的是来管理系统的风险，知道这些风险是什么，哪些风险是可接受的，以及你能够做什么来缓和风险。

我们称之为风险管理。我们会在本书的第 II 部分着重讨论风险管理，它是构建高可用系统的核心内容。

风险管理中的一个部分是风险缓和。风险缓和指的是当问题发生时，我们知道如何去尽可能降低问题所带来的影响。缓和意味着即使当服务和资源不可用时，依然尽可能确保你的系统以最好的、最完整的状态工作。风险缓和需要考虑哪些事情可能会出错，并且立即制订相应的计划，以便当问题发生时能够提供相应的解决方案。

示例2-1：风险缓和——一个没有搜索功能的网上商店

假设有一个售卖 T 恤的网上商店。它是一个很常见的在线商店，你可以在它的首页上浏览 T 恤，跳转到其他页面查看不同的 T 恤分类，并且可以搜索指定风格和类型的 T 恤。

为了实现搜索的功能，通常这类网上商店需要调用一个独立的搜索引擎，这可能是一个单独的服务，或者甚至由第三方的搜索服务提供。

但是，因为搜索功能是一个独立的功能，你的系统就存在搜索服务不可用的风险。你的风险管理计划需要确定出该问题，并且将“搜索引擎失败”列为风险之一。

如果没有风险缓和计划，当搜索服务失败时，可能会产生一个错误页面，或者返回不正确或无效的结果——不管怎样，它都会带来很差的用户体验。

这个示例中的风险缓和计划可能是这样的：

> 我们知道最受欢迎的T恤是红色条纹T恤，60%访问网站的用户最终都停留在（并很可能最后会购买）这个产品上。因此，如果搜索服务停止了，我们可以显示一个“很抱歉”的页面，下方是我们最受欢迎的T恤列表，其中就包括红色条纹T恤。这会鼓励遇到这个错误页面的用户，继续浏览别人曾经感兴趣的T恤。
>
> 12 此外，我们还可以显示一个“下一次购买享受10%折扣”的优惠券，这样就可以鼓励之前没有进行购买的用户，在搜索服务恢复正常后，继续回到我们的网站上进行购买。

示例2-1演示了什么是风险缓和，而确认风险、确定该如何处理风险，以及如何实现这些缓和措施的过程则被称为风险管理。

风险管理经常会暴露应用程序中未知的、需要立即修复的问题。它还可以用来处理已知的故障问题，减少故障恢复时间或者降低严重性。

可用性和风险管理息息相关。构建一个高可用的系统，主要就是要考虑如何管理风险。

要点4：监控可用性

除非你看到问题发生，否则你不会知道应用程序中存在着问题。你应当确保对应用程序进行了适当的监控，以便可以从外部和内部两个视角来观察应用程序的运行状况。

监控的程度取决于应用程序的特点和要求，但是通常必须具备以下这些监控。

服务器监控
: 监控服务器的健康状况，并且确保它们始终在有效运行。

配置变化监控
: 监控系统配置的变化，以便确定它们对应用程序的影响。

应用程序性能监控

深入了解你的应用程序和服务，确保它们按照预期运行。

人为测试

从用户的角度来实时检测应用程序的运行情况，以便在用户真正发现问题之前发现它们。

报警

当问题发生时通知相关人员，以便使问题可以得到快速有效的解决，将对用户的影响降低到最小。

如今市面上有许多非常优秀的监控系统，包括免费和付费的服务。我个人推荐 New Relic，它提供了之前提到的所有监控和报警能力。作为一款软件即服务（SaaS）的软件，它能够支持任何规模的应用系统的监控需求。[注2]

13

当你对应用程序和服务进行监控之后，请开始寻找它们的运行趋势。当你明确了一定的趋势之后，可以开始寻找一些异常值，将它们作为可能存在的可用性问题。你可以利用这些异常值，在系统发生故障之前通过监控工具来发送警报。除此之外，你还可以在系统增长过程中时刻进行跟踪，确保可伸缩性计划的实施。

你应当为服务间通信建立内部的、私有的运行目标，并持续对它们进行监控。通过这种方式，当出现任何与性能或者可用性相关的问题时，你都可以快速诊断出哪个服务或者系统出现了问题，并定位问题的原因。此外，你可以发现一些“热点”——即性能超出预期的地方，以及针对这些问题制订相应的开发计划。

要点 5：以预测和确定的方式来应对可用性问题

如果你对监控中所发生的问题置之不理，那么监控系统就毫无用处。这意味着当问题发生时必须要发出报警，这样你才能有所行动。除此之外，你应当建立整个团队都遵循的流程，帮助诊断问题，并轻松修复常见的故障问题。

例如，如果某个系统无法响应，你可能会有一系列措施来解决。这其中可能包括运行一个测试来诊断问题原因，重启一个已知会导致系统无法响应的守护进程，或者当其他手段都失败时重启整个服务器。为常见的故障问题提供标准化流程可以降低系统不可用的时间。此外，它们还可以提供更多有用的诊断信息，帮助工程师团队找到常见问题的根本原因。

注 2　我应该指出，虽然我为 New Relic 工作，但这不是我推荐它的原因。我在加入这家公司之前就已经在使用 New Relic 的工具。通过它们，我成功解决了应用程序性能和可用性问题，这也是我为什么开始选择在 New Relic 工作的原因。

当触发某个服务的报警时，该服务的负责人必须是第一个被通知到的。毕竟，他们负责
14 修复自己服务中的所有问题。但是，其他与之紧密相关或依赖的团队也应当收到报警信息。例如，如果某个团队使用了一个特殊服务，他们希望知道该服务什么时候出现故障，从而在问题发生时能够更加主动地保证自己的系统不受影响。

这些标准的流程和办法应当被写进支持手册中，团队中每个负责人人手一份。这本支持手册还应该包含相关系统和服务的联系人列表，以及当无法用简单手段解决问题时，需要上报问题的联系人列表。

所有这些流程、办法以及支持手册都应该提前准备好，以便当服务出现问题时，值班人员能准确知道如何在不同的情况下进行操作，快速恢复服务。这些流程和办法之所以非常有效，是因为故障通常都发生在一些不太方便的时间点，例如午夜或者周末这些效率比较低下的时间。这些建议可以帮助你的团队更聪明、更安全地将系统恢复到可运行状态。

做好准备

没人能够预测到可用性问题在什么地方、什么时间发生。但是你可以假设它们会发生，尤其是当你的系统面临越来越多的用户需求，变得越来越复杂的时候。提前做好处理可用性问题的准备，是降低问题出现概率和严重性的最佳方法。本章讨论的五个技巧，为保持应用程序的高可用性提供了可靠有效的手段。

第3章 测量可用性

测量可用性对保证系统高可用非常重要。只有通过测量可用性，你才能了解应用程序当前的运行状况，以及检查可用性随时间发生的变化。

测量 Web 应用程序的可用性，最常用的办法是计算用户可访问的时间百分比。我们可以通过以下公式来计算一段时间内的可用性：

网站可用性百分比 =（该期间的总秒数 - 系统宕机的秒数）/ 该期间的总秒数

我们举一个示例来说明。假设在整个 4 月，你的网站宕机了两次，第一次宕机 37 分钟，第二次宕机 15 分钟。那么你的网站的可用性是多少呢？

示例3-1：可用性百分比

系统宕机的总秒数 =（37 + 15 ）* 60 = 3, 120 秒

该月份的总秒数 =30 天 * 86, 400 秒 / 天 = 2, 592, 000 秒

网站可用性百分比 =（该期间的总秒数 - 系统宕机的秒数）/ 该期间的总秒数

网站可用性百分比 =（2, 592, 000 秒 -3, 120 秒）/2, 592, 000 秒

网站可用性百分比 =99.8795

因此，你的网站的可用性为 99.8795%。

从这个示例中可以看出，只要很短一段故障时间，就会对可用性百分比造成影响。

N 个 9

通常我们会用“N 个 9”来形容可用性。这是表示高可用性百分比的一个简化方式。表 3-1 列举了各个 9 的含义。

表3-1: N个9

N 个 9	百分比	每月的故障时间*
2 个 9	99%	432 分钟
3 个 9	99.9%	43 分钟
4 个 9	99.99%	4 分钟
5 个 9	99.999%	26 秒
6 个 9	99.9999%	2.6 秒

* 假设每个月有 30 天，每月共计 43,200 秒。

在示例 3-1 中，我们可以看到网站可用性百分比小于 3 个 9（比较 99.8795% 和 99.9%）。对于一个要维持 5 个 9 的网站来说，每个月只能宕机 26 秒。

什么样的可用性是合理的

如果你希望系统是高可用的，那么什么样的可用性数值是合理的呢？

对于这个问题，没有唯一的答案，因为这取决于你的网站、用户期望、业务需要以及业务期望。你需要自己来决定什么样的数字能够满足你的业务。

通常，对于简单的 Web 应用程序来说，3 个 9 的高可用性是可接受的。通过表 3-1 可以看到，这表示每个月的故障时间不能超过 43 分钟。而对于一个高可用的 Web 应用程序来说，一般会采用 5 个 9。这意味着每个月的故障时间只有不到 26 秒。

17

不要上当

不要想当然地以为自己的网站是高可用的，计划中和日常维护工作所导致的系统不可用时间也要计算在可用性百分比之中。

我经常会听到这样的评论：“我们的应用程序从来也没有出现过故障。因为我们会经常性地进行系统维护。通过每周两个小时维护时间窗口，我们保证了系统的高可用性。”

这真的能保证应用程序的可用性吗？

让我们来计算一下。

示例3-2：维护窗口示例的可用性

网站可用性百分比 =（该期间的总秒数 - 系统宕机的秒数）/ 该期间的总秒数

每周的小时数 =7 天 * 24 小时 = 168 小时

每周不可用的小时数 =2 小时

网站可用性（没有故障）=（168 小时 -2 小时）/ 168 小时 = 98.8%

网站可用性（没有故障）=98.8%

即使应用程序没有出现任何故障，最好也只能达到 98.8% 的可用性。这甚至低于 2 个 9 的标准（比较 98.8% 和 99%）。

计划中的维护和未预期的故障效果其实差不多。你的用户希望应用程序是可用的，但是如果不可用，你的用户就会有负面的体验。他们不会关心你这是计划中的维护还是意外的故障。

通过数字来体现可用性

测量可用性对于保证系统高可用十分重要，不管是现在还是将来。本章讨论了测量可用性的常用方法，并提供了一些如何选择合理的可用性的建议。

19

第4章

提高下降的可用性

你的应用程序在线上运转正常，系统一如往常，团队高效运转。所有事情看上去都是那么好。你的流量持续增长，销售部门非常高兴。所有一切都很好。

然后出现了一些小意外。你的系统发生了一次预料之外的故障。但是这样还好，你的可用性到目前为止还是非常好。一次故障并不是什么大问题。你的流量还在持续增加。每个人都对此不以为然——这只是“一件小事”罢了。

然后另一次故障又再次发生了。好吧，总体上来说，我们还是干得不错的。不用慌张，这只不过是另外“一件小事”罢了。

然后又出现了一次故障……

现在你的CEO开始有一点担心了。用户开始询问出了什么事情。你的销售团队开始担心了。

然后又出现了一次故障……

突然，你曾经认为稳定且正常的系统正在变得越来越不稳定，你的故障正在有越来越多的人关注。

现在你遇到真的麻烦了。

发生了什么事？保持系统高可用是一件很难的事情。可用性开始下降时，你应该做什么？你的应用程序可用性已经下降或者正在开始下降，而你需要提高它来保证让用户满意，你应该做什么？

当可用性下降时，知道如何处理会帮助你避免陷入恶性循环。以下几节描述了当可用性开始下降时你可以采取的几个步骤。

测试并跟踪当前的可用性

要理解可用性发生的变化，你必须首先测量出当前的可用性是多少。通过跟踪系统可用及不可用的时间，计算出一个可用性百分比，可以帮助你了解一段时间内的可用性程度。通过这个值，你可以判断出可用性是在提高还是降低。

你应当不断地监控可用性百分比，并定期收集结果。此外，你需要覆盖系统中所有的关键事件点，例如更改系统配置或者升级的时候，这样可以了解到系统事件和可用性问题之间的关系，也能够帮你确定系统的可用性风险。

如果你需要复习如何测量可用性，可以参考第 3 章。

接下来，你必须从可用性的角度出发，来理解系统应当如何运行。服务分级是一个可以帮助你管理应用程序可用性的工具。服务分级指的是为各个服务打上一些简单的标签，表示该服务对于系统的关键程度。这使得你和团队能够区分出哪些是至关重要的服务，哪些是很重要但非必需的服务。我们将在第 16 章来深入讨论服务分级。

最终，你需要创建并维护一个风险模型。通过这个工具，你可以了解当前的技术债务以及相关的风险。我们会在第 7 章来更加完整地介绍风险模型，以及在第 5 章和第 6 章对风险进行初步讨论。

既然你已经知道了如何跟踪可用性，以及如何确定并管理你的风险，你需要定期审查你的风险管理计划。

除此之外，你应该建立并实施风险缓和计划来降低系统的风险。它可以为你和开发团队提供一系列具体的实施任务，解决掉系统中最危险的部分。我们会在第 8 章来详细讨论这部分内容。

将手动流程自动化

为了维护高可用性，你需要移除未知以及不可控的因素，而执行手动操作就是一个常见的为系统带来不可控或未知结果的原因。

你应该永远不要在生产环境上执行手动操作。

当你对系统进行了某项更改后，它对系统造成的影响可能有益，也可能有害。因此，仅

允许使用可重复性的任务会给你带来如下的好处：

- 在实施更改前进行测试的能力。在更改之前进行测试，对于避免因为失误造成的故障来说至关重要。
- 完全控制任务执行的能力。这使得你可以在进行更改之前，对更改的内容和步骤进行完善。
- 由第三方审查更改任务内容的能力。这降低了更改任务产生未预期结果的可能性。
- 通过版本控制来管理任务的能力。版本控制系统可以让你清楚了解任务被更改的时间、人员以及原因。
- 对相关资源进行更改的能力。通过更改来提高一台服务器的能力是很不错，但是如果能够将相同的更改，完全一样地应用到所有受影响的服务器上，能够为我们带来更大的价值。
- 让所有相关资源保持一致的能力。如果你总是对资源（例如服务器）进行临时的手动更改，那么它们之间会逐渐产生不同的行为方式。由于无法找到可比较的基准行为，所以这会增加诊断问题的难度。
- 实施重复性任务的能力。重复性的任务都是可审计的任务。可审计的任务指的是你可以事后从整个系统层面分析任务的影响是积极的或是消极的。

事实上，有很多系统的生产环境没有任何人有权访问。唯一能够访问的方式就是通过自动化的程序。正是出于我们以上提到的几点原因，这些系统的管理员才将系统隔离了起来。

总而言之，如果你不能重复执行一个任务，那么这个任务就没什么用。事实证明，自动化执行更改可以保持系统和应用程序的稳定性。这其中包括服务器配置更改、性能调优、重启服务器、重启任务、改变路由规则，以及升级和部署软件包。

自动化部署

通过自动化部署，你可以确保整个系统中的更改都是一致的，并且下次再进行相似更改时，你可以知道它所带来的影响。除此之外，自动化部署系统可以帮助你更可靠地将系统回滚到已知的良好状态。

配置管理

不要对服务器的内容“临时手动地更改某个配置变量”，而要使用自动化的方式来进行更改。例如，你可以编写一段进行更改的脚本，然后通过你的软件更改管理系统来检查该脚本。这样，你可以对系统中的所有服务器进行统一更改。此外，当需要向系统中添加新的服务器或者替换旧服务器时，使用已有的配置可以帮助你更安全地将新服务器添

加到系统中，将影响降低到最小。像 Puppet 和 Chef 这类工具可以帮助你更容易地管理整个流程。

不只是服务器，这对于基础设施组件也是一样的。其中包括交换机、路由器、网络组件以及监控程序和系统。

为了让配置管理能够发挥作用，必须坚持使用配置管理流程来进行所有的系统更改。不管在任何情况下，都绝对不能越过配置管理系统来手动进行更改。切记。

不要担心，我搞定了

你可能都想象不到，我接收到多少封像这样的运维更新邮件："我们的服务器昨天晚上遇到了一个问题。我们遇到了操作系统最大打开文件数量的限制，于是我手动修改了系统变量，并增加了可允许打开的最大文件数量，服务器已经恢复正常。"

这意味着，一旦有人不小心覆盖了之前的更改，服务又会出现问题，因为没有任何文档记录了这次更改。或者，其他运行相同程序的服务器也会遇到这个问题，因为它们没有进行这个更改。

一致性、可重复性以及对细节的专注，是成功执行配置管理流程的关键因素。我们这里所描述的标准的、可重复的配置管理流程，对于让你的大规模系统保持高可用性至关重要。

更改实验和高频次更改

拥有一个高可重复的、高自动化的更改和升级流程，带来的另一个好处是允许你对更改进行实验。假设你需要对服务器进行一次更改，并且你相信这会提高应用程序的性能（例如在"不要担心，我搞定了"中介绍的打开文件最大数量）。通过自动化的配置管理流程，你可以做到以下几点：

23

- 用文档记录你想要进行的更改。
- 让经验丰富的人来审查这次变更，可以提出建议和改进。
- 在一个测试或者预发布环境上进行测试。
- 快速、简单地部署更改内容。
- 快速检查结果。如果这次更改没有达到想要的结果，你可以快速回滚到上一次的正常状态。

实现这个流程的关键是，拥有一个具有回滚能力的自动变更流程，以及对系统频繁、轻

易进行小范围更改的能力。[注1] 前者可以让你对多台服务器进行统一更改，后者可以让你对更改内容进行实验，并且在失败的时候进行回滚，让用户几乎感觉不到影响。

自动化的变更完备性测试

当拥有自动化的变更和部署流程时，[注2] 你可以对所有更改实现一个自动化的完备性测试。你可以使用浏览器测试程序来测试 Web 应用程序，或者使用像 New Relic Synthetics 这样的测试程序来模拟用户的交互。

当准备好将变更内容部署到生产环境时，你可以让部署系统先将它们自动部署到某个测试或者预发布环境上。然后，就可以运行这些自动化测试，并检验变更内容的正确性了。

如果所有测试通过，你可以将变更内容按照统一的方式部署到生产环境中。根据测试的不同情况，你应该定期对生产环境进行测试，并验证变更的正确性。

通过让整个流程自动化，你会对每次更改更加有信心，知道它们不会对生产系统造成负面的影响。

24 改进你的系统

现在，你已经拥有了一个可以监控可用性的系统，一个可以跟踪风险和缓和风险的方法，以及一个简单、安全、统一进行变更的流程，你可以将精力集中到如何提高应用程序的可用性这一点上了。

你需要经常检查你的风险模型（我们在本章已经介绍过，会在第 7 章再详细介绍）以及恢复计划，将它们作为你问题复盘流程中的一个部分。执行那些能够降低风险模型中风险的计划。将这些变更通过自动化、安全的方式进行部署，并进行完备性测试。之后，你应当检查这些变更对可用性的提升效果。持续这个流程直到可用性达到你希望并且应该达到的程度。

你可以在第 13 章中了解到如何从故障服务中进行恢复。

将可用性指标公开给其他成员和团队，会帮助大家了解你对系统可用性的提升。

不断变化和发展中的应用程序

随着系统的发展，你需要处理越来越大的流量和越来越多的数据。这些流量以及数据的

注 1　根据 Amazon CTO Werner Vogels 所述，Amazon 在 2014 年对个人主机进行了 5000 万次部署，平均一秒一次。

注 2　这个流程可以是但不一定必须是流行的持续集成和持续部署（CI/CD）流程。

增长也会带来更多的可用性问题。第Ⅳ部分将介绍关于应用程序如何伸缩的更多内容，其中许多话题都会帮助到正在面临可用性问题的程序。尤其是在第 14 章讨论的如何管理失误和错误，第 18 章中讨论的服务等级协议（SLA）管理，以及第 16 章和第 17 章中讨论的用来确定可用性关键服务的服务分级制度。

在第 9 章中，我们将进一步讨论如何实现“比赛日”测试，用它来测量应用程序在不同故障模式下的执行情况。

时刻关注可用性

通常来说，你的应用程序会不断发生变化。因此，你的风险管理、风险缓和、应急方案以及恢复计划也需要不断进行相应的改变。

当可用性开始下降时知道该如何处理，可以帮助你避免进入恶性循环。本章中所介绍的内容可以帮助你管理应用程序，也可以帮助你的团队避免走入误区，保持系统的高可用性。

第II部分 25

风险管理

如果你无法识别系统中的风险，你就不可能管理风险。

……但是，同样存在“未知的未知”——即我们不知道自己不知道。如果纵观我们国家和其他自由国家的历史，往往困难的是后者。

——唐纳德·拉姆斯菲尔德（美国前国防部长）

27

第 5 章

什么是风险管理

所有的复杂系统都存在风险，它是所有系统不可避免的一个部分。我们不可能消除一个复杂系统（例如一个 Web 应用程序）中的所有风险。但是，检查风险并确定风险的接受程度，对于保持系统健康非常重要。

我们在本章会整体介绍什么是风险，以及如何来识别风险。然后，我们会介绍风险管理流程，其可帮助降低风险所带来的影响。

我们先来看一下示例 5-1，其中再次引用了第 1 章中的重大赛事示例。

示例5-1：重大赛事的风险管理

这里再简单回顾一下之前介绍过的重大赛事示例：周日有一个重大的赛事，你邀请朋友们来到家里，一起在新买的电视上观看比赛。比赛马上就要开始了，但是突然停电了，灯和电视都灭了。这场比赛对于你和朋友们来说也结束了。你给电力公司致电，他们说“我们非常抱歉，我们只能保证电网 95% 的可用性”。

示例中的电力公司就存在着风险，他们承受着重大赛事时停电的风险。

他们甚至还将风险进行了量化（只保证大约 95% 的电力）。

电力公司知道哪些事件会导致停电，例如电线损坏。因此，为了保证电线的安全，他们通常会这么做：

28

- 将电线埋到地下（为了避免被风吹走）。
- 加固电线（为了降低被风暴吹断的几率）。
- 增加备用电力系统（即使一根电线损坏，另一条还可以继续工作）。

但是这些方案都是有成本的。是否值得投入资金加固电线？是否值得将电线埋入地下？

相比风险所带来的损失，这些降低风险的投入是否值得？这些问题都属于风险管理问题，也是我们本章要讨论的问题。

管理风险

风险管理包括确定系统中风险的位置，确定哪些风险必须消除，而哪些可以暂时存在，以及如何降低这些存留风险发生的可能性和严重性。

当某个风险被触发（或者发生）时，你或你的系统会遭受损失。这些损失可能包括公司或者用户的数据丢失，无法给用户提供服务，或者是返回无效或错误的结果。不管怎样，最终这些损失都会让用户对你管理他们数据和业务的能力丧失信心，最终承受损失的只能是你自己。

但是，你必须从辩证的角度来评估这样的损失：是否投入的成本可以避免风险再次发生？

最终，风险管理就是在消除风险的成本与风险发生的成本之间，保持平衡。

识别风险

风险管理的第一步就是列出所有已知的风险，以及它们的危害性和发生的可能性。

我们称这个列表为风险模型，图 5-1 展示了一个风险模型的例子。

Risk ID	System	Owner	Description	Date Identifie	Likelihoo	Severi	Mitigation Plan	Status	ET	Monitoring	Triggered Plan(what to do if risk occurs?)	Comments
1	FrontEnd	FE Dev Team	Requires User identity service to function. Front end fails if service is down.	10/13/15	Low	High	Perhaps we can cache the date for a period of time?	Open	5/26/16	Yes	If it happens often, we may have to look into improving independence on UI service.	

图5-1：风险模型示例

最初的风险模型的创建需要团队的头脑风暴。你可以从以下多个来源收集风险模型：

29

- 开发人员的头脑风暴
- 已知的重点售后支持问题
- 已知的安全风险和漏洞
- 已知的系统不完善或者缺失的能力

- 已知的性能瓶颈点
- 已知的流量峰值和变化
- 来自业务负责人、支持人员或者用户的特殊考虑
- 已知的系统技术债务

在这个列表里有一些显而易见的事项，也有一些你可能从未想到的事项。这是件好事，因为你应该尽可能地去发现更多的风险威胁，如果你认为自己都考虑到了，有可能说明你还没有考虑得足够细致。

建立风险模型还包括为风险出现的可能性和造成的影响分配优先级。

我们会在第 7 章详细讨论这个列表中的事项。

消除最严重的风险

在你完成这个列表之后，应该从中确定出最严重的风险事项。你如何知道哪些风险是最严重的呢？可以先找出经常会发生的风险，或者还没有发生但一旦发生就会造成严重问题的风险。当然，最严重的风险就是既经常发生又对系统有严重影响的风险。我们会在第 6 章来讨论严重性与可能性的区别，以及如何利用这一点来帮助你找到最严重的风险，从而更好地管理风险。

在图 5-1 中，“如果用户身份服务停止服务会导致前端无法展现”可能就是我们最严重的风险了。

一旦你识别出了高危风险，请将它们加到风险计划中，并定期检查它们是否依然存在。

风险缓和

对于所有的风险，不管它们是不是最危险的，我们都应当尽可能地去思考有没有降低发生风险几率或者影响程度的办法。这些办法被称为*风险缓和措施*。

风险缓和措施的价值很高，你需要重点发现那些既能够降低风险（不管是严重性还是发生几率）又易于实现或成本合理的措施。

我们以图 5-1 中的“如果用户身份服务停止服务会导致前端无法展现”的风险为例，一个可能的风险缓和措施是缓存某些用户身份信息，这样即使用户身份服务不可用，前台也可以将这些信息显示出来。

你可以重点关注那些最严重的风险，寻找降低这些风险的方式。但是，你也要注意那些

不太容易被快速修复的风险，找到降低这些风险严重性和发生几率的办法，几乎就等同于修复它们。

定期检查

如果你不定期检查风险模型，它就会慢慢失去作用。你的团队应当至少每个季度都对风险模型进行检查，但是对于非常活跃以及极其关键的系统可能需要每月检查。此外，在每次出现事故之后也应该重新检查风险模型，是否存在已知风险能够正确覆盖所发生的事故？

当你检查风险模型时，首先要检查最近新增或者确定的风险，并将它们添加为新的事项，同时移除已经不再存在的风险事项。

接下来，你需要检查风险的严重性或者可能性的变化。通常，风险缓和措施会帮助降低风险的严重性和可能性，或者你了解到更多可能会增加风险发生几率或者严重性的信息。我们经常会遇到，在进行完上一次检查之后，风险却发生了，这时候你可能会将之前低优先级的风险标记为高优先级。这时候，你需要考虑将它们加入到风险模型中是否可以消除（修复）这些风险？

最后，检查是否有新的或者更新后的风险事项需要被加入到模型中。

对风险管理的总结

你如何来管理系统中的风险？以下是一些你可以用来参考的基本步骤。

识别风险
: 首先，创建一个系统中已知风险的列表，即风险模型。对列表中各个风险事项排列优先级。

消除最严重的风险
: 找出列表中最大的风险，制订相应的解决计划。

31

风险缓和
: 对于无法暂时解决的主要风险，制订一个缓和计划来降低风险发生的可能性和严重性。

定期检查
: 定期检查你的风险模型。

33

第6章

可能性与严重性

理解严重性和可能性之间的关系非常重要。管理风险，你需要知道何时该考虑严重性而不是可能性，或者相反。理解两者的区别对分析系统中风险的重要性至关重要。

我们认为所有风险都由以下两方面组成。

严重性
: 如果发生风险，所需付出的代价（例如，如果停电，用户会受到什么影响）。

可能性
: 风险发生的几率（例如，巨型风暴发生的几率）。

管理风险就是管理这两者，你可以降低风险的严重性或者可能性。对于所有已知的风险，你不需要双管齐下，但是同时考虑这两者对提高管理风险的能力非常重要。

风险的重要程度就是风险发生的严重性与可能性两者之和。如果要成功地管理风险，你必须同时考虑两者以及它们之间的关系。为了降低风险，你需要至少降低其中之一，或者严重性，或者可能性。

理解二者区别的最佳办法，就是通过观察各种各样的风险，了解它们各自在可能性和严重性上的不同。我们将通过如下示例来了解二者之间的区别。

34 假设我们正在管理一个在线的 T 恤商店。这个商店是一个标准的在线零售商店，提供当前在售的 T 恤列表，每个 T 恤的详细情况页面（包括展示图片），以及一个用户用来购买、支付和快递的订单处理系统。

现在我们来看一下这个在线商店有哪些常见风险。

10 佳列表：低可能性，低严重性

在使用这个例子之前，我们先假设它有一个展示售卖量前十的 T 恤的列表。访问网站的用户可以看到这些销售量最好的 T 恤，快速、方便地查看商品详情并进行购买。

现在，如果这个 10 佳列表由于一些原因（也许是服务故障）无法展示，会发生什么呢？如果它无法展示，我们可以将它替换成一个静态的 T 恤列表，但是这些 T 恤不需要必须是卖得最好的前十个款式。这个服务很少出现故障，因为 10 佳列表很好生成，不应该出现什么问题。

那么关于这个 10 佳列表的显示，它的风险是什么呢？

我们来看一看这个风险：

- 风险的可能性很低，因为展示列表的服务非常稳定（我已经说过，这个列表很容易生成）。
- 但是如果列表没有显示，问题的严重性有多大呢？我已经提到，如果 10 佳列表没有显示，会显示一个代替的列表。虽然不够理想，但是它对用户的影响很小，并且对业务的影响也不会太大。因此，这个风险的严重性也是较低的。
- 所以，这是一个低 / 低的风险，意味着风险的可能性和严重性都很低。

像这样的风险很容易被忽视，通常也不会受到进一步关注，因为它很少发生，即使发生影响范围也很小。

订单数据库：低可能性，高严重性

在这个例子中，我们假设订单数据存储在传统的关系型数据库中。当用户生成某个订单时，会在数据库中增加一条记录；而当你处理、收费以及快递订单时，会更新数据库中的数据。随后，你会用这些数据来生成财务报表，为商业计划或者税务计算等商业用途提供帮助。

35

因为数据库很重要，所以你采用了带有备份功能（例如 RAID 磁盘阵列）的高质量硬件环境。你还经常对数据进行备份。

但是，数据库仍然是一个故障单点。它包含了大量关键的商业数据，如果数据库不可用，则网站也无法提供服务（无法提交任何订单）。数据库故障会带来极大的损失。

那么关于这个订单处理系统的数据库，它的风险是什么呢？

我们来看一看这个风险：

- 失败的可能性很低，因为我们使用了高质量的、带有备份的硬件。数据库本身非常可靠。
- 但是，数据库故障的严重性可能很高。因为如果数据库真的发生故障，整个订单处理流水线就无法运作，你会面临丢失关键商业数据的风险。
- 因此，这是一个低 / 高的风险，意味着它的可能性很低但是严重性很高。
- 像这样的风险很容易被忽视，因为它们很少发生（可能性低）。但是，忽视它们可能会带来严重的后果（严重性高）。

考虑到高严重性，你也许会希望采取措施来降低这个风险的严重性。例如，你希望使用一个从数据库，作为主数据库的实时副本，这样当主数据库发生故障时，可以快速将从数据库作为主数据库使用，避免大量时间和数据上的恢复工作。另外，你还可以采用分布式数据库，将数据分散到多台服务器上，这样即使其中一个数据库服务器发生故障，整个集群还可以提供服务。

不管使用上面提到的哪种方法，都可以将这个风险从低 / 高风险降低到低 / 中风险（低可能性，中严重性），甚至是低 / 低风险（低可能性，低严重性）。

我们会在第 8 章来进一步讨论，像这样能够显著降低风险严重性的风险缓和措施。

自定义字体：高可能性，低严重性

在这个 T 恤在线商店的示例中，假设你需要对所有文字和描述使用自定义字体，以便让网站看上去更加简洁漂亮。你找到了理想的字体，不过它由第三方的字体服务方提供。为了使用这个字体，你会让用户的 Web 浏览器直接从第三方下载字体文件。如果自定义字体无法使用，浏览器会使用之前标准的系统字体。

36

但是，你注意到字体服务方有时会发生问题，且过于频繁。当字体服务方出现问题时，你的用户就无法使用好看的自定义字体了。

不幸的是，这经常发生。

那么关于好看的自定义字体，它的风险是什么呢？

让我们来看一看这个风险：

- 字体不能显示的可能性很高，因为字体服务方经常会发生问题。
- 但是，当问题发生时，你的网站可以继续提供服务——只是看上去没有你想要的效果。因此，这个问题的严重性较低。

- 你的网站可能会看上去不那么美观，但是没有影响任何功能。
- 因此，这是一个高 / 低风险，意味着它发生的可能性高但是严重性低。

缓和这个风险的方法主要是降低问题发生的可能性。你可以与第三方一起来改进服务的可用性，降低问题的发生概率。或者，你可以挑选出多个可以提供同一或类似字体的备份提供方，如果第一个提供方出现故障，则切换到其他提供方。这些都是你可以用来降低风险发生概率的手段。

由于该风险的严重性已经很低，没有必要再降低其严重性。

T 恤图片：高可能性，高严重性

在这个 T 恤在线商店的示例中，我们来看一看网站上展现的 T 恤图片。因为用户如果无法看见 T 恤是什么样子的，是不会进行购买的，所以这些图片对于商店来说非常重要。如果网站无法显示出 T 恤图片，用户会选择离开，你也会损失订单。

但是，你用来存储图片的服务器却非常脆弱。它时常出现故障，并且似乎在磁盘读取图片方面存在问题。这台服务器很旧，需要被替换成新的服务器。它经常死机，需要定期重新启动，也经常由于需要更换部件而停机。但是不管怎样，你还是使用这台服务器来存储图片。

37

那么关于图片无法显示导致网站无法使用，它的风险是什么呢？

让我们来看一看这个风险：

- 图片无法显示的可能性很高，因为服务器不稳定并且经常出现故障。
- 风险的严重性也很高，因为如果图片无法显示，用户会离开网站，也不会购买商品。
- 因此，这是一个高 / 高风险，意味着它的可能性很高（硬件经常故障）并且严重性也很高（用户不会进行购买）。

这种类型的风险最为严重，因为它发生的概率很高，并且会对业务产生严重的影响。

你最应该关注这一类风险。

这个示例可能看上去显而易见，但是应用程序中会存在很多类似的高 / 高风险。不过，通常这类风险都隐藏得很深，你只有仔细审视系统才会发现它们，这也是为什么说风险管理如此重要。

39

第 7 章

风险模型

管理风险的第一步是理解系统中已有的风险。识别、标记并对已知的风险排列优先级，这就是风险模型所要做的事情。

我们在第 5 章中已经介绍过了，风险模型是管理系统中风险的一个关键方面。它是一张包含了系统所有已知风险的当前状态的表格。

图 7-1 列举了一个风险模型的示例。

Risk ID	System	Owner	Description	Date Identified	Likelihood	Severity	Mitigation Plan	Status	ETA	Monitoring	Triggered Plan	Comments
1	FrontEnd	FE Dev Team	Requires User Identity service to function. Front end fails if service is down.	10/13/15	Low	High	Perhaps we can cache the date for a period of time?	Open	5/26/16	Yes	If it happens often, we may have to look into improving independence on UI service.	

图7-1：风险模型示例

模型中的每一行都表示系统中一个独立的、可量化的风险。表格中的各列包含了具体风险项的详细描述。

对于每个风险项，风险模型包含了以下内容。

40 风险 ID

这是每个风险的唯一标识。它可以是任何类型的，但通常选择唯一的整数标识，这是最简单的方式，也能够满足需要。[注 1]

注 1　但是这个 ID 不应该是表格中的行号。因为模型中的行可能会进行排序、新增或者删除，这些都会改变某些风险在模型中的行号。风险 ID 应该是在跟踪风险的整个过程中不会发生变化的唯一标识。

系统

这是包含风险的系统、子系统或者模块的名称。这个值取决于你对应用程序的定义和划分，通常会是像“前端”、“主数据库”、“服务 A”类似的值。

所有人

负责该风险的个人（或者团队）的名称，同时也负责制订该风险的缓和计划和解决计划。

风险描述

该风险的概要描述。它应该尽可能简短，便于查看，但又不能太短，以便唯一、准确地标识该风险。

标识日期

识别该风险并将其添加到模型中的日期。

可能性

表示该风险发生的可能性（分低、中、高三级）。我们在第 6 章中详细讨论了这个值，你可以使用这个值对风险模型进行排序，确定哪些是最需要关注和立即解决的风险。[注 2]

严重性

这表示风险发生的严重性或者影响（分低、中、高三级）。我们在第 6 章中详细讨论了这个值，你可以使用这个值对风险模型进行排序，确定哪些是最需要关注和立即解决的风险。

风险缓和计划

这一列描述了可以使用的或者正在使用的，用来降低该风险严重性或者可能性的风险缓和措施。

状态

41

该列表示该风险的当前状态。这个值通常是“活跃”、“已缓和”、“正在修复”或者“已解决”等内容。

ETA

表示该风险距离计划解决日期（如果有的话）的估计时间。

监控

该列表示你是否对该风险的发生进行了监控，如果是，则包含你实现监控的步骤。如果你没有监控该风险，应该标明原因，以及计划对其进行监控的估计时间。

注 2　为了能够方便地对可能性和严重性的值进行排序，你可以使用 1-3 来代表由低到高的程度。通常，我们会按照“1- 低”、“2- 中”、“3- 高”的规则，并利用表格编程的能力来限制该值只能允许这三个选择。

触发计划

该列表示如果该风险真的发生，你计划如何来处理它。触发计划通常是一个管理层面的计划，而不是一个事件 - 响应的计划。[注3]

备注

使用该列来记录任何不适合记录在风险描述中，或者不属于风险描述的信息。

除此之外，你还可以根据公司情况，在模型中添加你认为重要的列，例如：

跟踪 ID

如果你通过一个缺陷跟踪系统或里程碑跟踪系统来管理风险，可以用该列来记录该风险在系统中的跟踪 ID。

历史

该风险在过去是否已经发生过？什么时间？发生频率？等等。

风险模型的作用域

现在，你可能在想“我应该为整个公司建立一个风险模型，还是应该分别为每个团队或者服务建立一个风险模型？”

这是个好问题。对于小型公司来说，整个公司可以使用一个风险模型，但是它很快会变得相当笨重。每个服务一个风险模型可以在服务层面提供良好的可见性，但是降低了公司层面的可见性。像“哪个服务的风险对于公司最重要？”这样的问题就变得难以回答。

我推荐为每个团队提供一个风险模型。因为通常由团队来决定要完成的功能或问题，以及它们的优先级，所以也应当从团队层面来管理风险模型，由团队来决定其中各项风险的优先级。你可以在第 15 章了解更多从团队层面来管理风险的内容。

42

最后，你应当根据组织架构来设定风险模型的作用域。例如，每个团队、小组或者组织各自使用一个风险模型，来管理自己的工作范围和优先级。他们可能从上级管理部门接受任务和指导，但是会自己来决定大多数工作的优先级和执行内容。

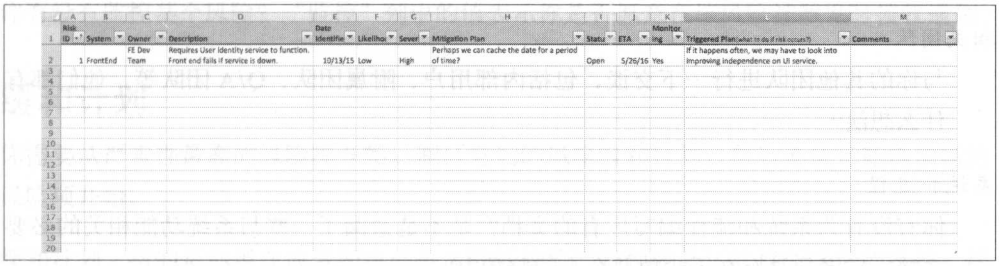
创建风险模型

首先，你需要借助于一个风险模型模板。我们已经为你创建了许多流行的表格工具的模板，你可以在我们的网站（*http://bit.ly/architectbookdl*）上下载它们。虽然你可以随意修改它们，但是对于第一个风险模型来说，最好还是尽量不要修改。当你在使用模型和管

注 3　事件响应计划应该是已经制订好的，值班人员在事件应对手册或者其他工具中可以找到它们。

理风险方面有了一些经验之后，可以再将模板修改成你想要的样子。

为了向你说明如何使用它，这个模板中提供了一条风险的示例，如图 7-2 所示。你可以在开始之前删掉这条风险。



Risk ID	System	Owner	Description	Date Identified	Likelihood	Severity	Mitigation Plan	Status	ETA	Monitoring	Triggered Plan(what to do if risk occurs?)	Comments
1	FrontEnd	FE Dev Team	Requires User Identity service to function. Front end fails if service is down.	10/13/15	Low	High	Perhaps we can cache the date for a period of time?	Open	5/26/16	Yes	If it happens often, we may have to look into improving independence on UI service.	

图 7-2：下载的风险模型

通过头脑风暴建立风险列表

当你准备好模板后，第一步是通过头脑风暴得到一份风险列表。你应当试着把所有你能想到的风险都列出来，而不仅仅是那些你关心的风险。在这个过程中不要去分析它们，只要把它们尽可能都列出来就可以了。

以下是一些可以进行头脑风暴的很好来源。

开发团队

与你的开发团队召开一个会议。团队成员会对他们的服务有大量的担忧，仔细聆听他们的担忧，并将它们作为风险项添加到列表中。

售后支持

看一下你的售后支持情况。是否存在有高于正常支持压力的问题？你的支持人员都在说什么？你有没有可以查看的支持论坛？需要大量支持的方面通常都会产生系统风险。

安全威胁

思考一下已知的安全威胁和漏洞。不管它们有多么严重或重要，它们都是服务所要面临的风险。

待完成的功能列表

浏览一下你待完成的功能列表，是否存在系统健康所缺失的关键能力？尤其是仔细检查与监控和维护有关的功能。

性能

思考一下系统的性能情况。有没有你注意到的性能较差的地方？

业务负责人

与业务负责人进行一下交谈。他们有什么顾虑？

相关团队

与你的其他团队进行一下交谈，包括内部用户、附属团队、Q/A 团队等。他们都有什么想法？

系统和流程

你有没有为系统和流程编写应有的文档？是不是遗漏了一些与系统功能相关的必要文档，或者只是存在于少数几个人的脑袋中？

技术债务

你的团队中有没有已知的、明确的技术债务？这些技术债务包括难以理解的、过于复杂的，或者过于冗余的代码。已知的技术债务通常都会是风险项。

你很可能发现列表中有些显而易见的风险项，但也会有些令你感到惊讶的项目。这很好，你会希望发掘尽可能多的风险漏洞，如果没有任何令你惊讶的风险项出现在列表中，有可能说明你考虑得还不够多。

填写可能性和严重性字段

现在，遍历整个列表，填写每一项风险的可能性和严重性字段。对于这两个字段，可以使用低 / 中 / 高（或者类似）的值。

44

确保你能够准确区分可能性和严重性的概念。如果有任何不清楚的地方，可以重新阅读第 6 章。通常，在这个步骤中很容易将可能性和严重性搞混。

先填写可能性，再填写严重性，这样可能会更容易一些。记住，风险项一旦发生会非常严重、但是几乎不可能发生的情况非常常见（同样，也可能非常容易发生，但是发生后却没有什么严重影响）。最终，你会得到许多状态为 H/H、H/L、L/H 和 L/L 的风险项，这是正常的。

不管你如何来完成它，如果你混淆了可能性和严重性，那么这份列表也就失去了它应有的意义。

与你的开发团队进行另一次头脑风暴有助于完成这个任务。这次头脑风暴应该与之前提到的分别进行，不要在识别风险的头脑风暴中同时分析可能性和严重性。

风险项详情

现在，你可以填写风险模型中其他的基本信息，包括系统、所有人、日期和状态列。

确认你为每个风险项都分配了一个风险 ID（一个从 1 开始的数字即可）。

你有没有监控某个风险？“监控”列中的值表示你是否愿意在该风险被触发时接收通知。

缓和计划

你需要从严重性最高的风险项开始，制订它们的风险缓和计划。然后再从可能性最高的风险项开始。

缓和计划，指的是你现在或者即将准备为降低风险严重性或者可能性所采取的措施。缓和计划并不是要消除风险，它只是降低风险的严重性和可能性。

当你执行完缓和计划中指定的方案后，风险的严重性或者可能性应该发生下降，这时可以将该缓和计划删除。如果需要，可以增加新的缓和计划。

你不需要为风险模型中的每个风险项都制订缓和计划。有些风险项是明显必须被解决的，不能只是降低可能性或者严重性。此外，有些低可能性、低严重性的风险也不需要再为其制订缓和计划。

触发计划

45

触发计划指的是当风险真正发生时你需要采取的措施。这可能会像“修复缺陷”这样简单的描述，也可能是更详细的描述。例如，如果发生了某个风险，是否有一些可以立即执行并降低影响的措施？如果有，它们也应该被加入到触发计划中。

从严重性最高的风险项开始，为每个风险项制订相应的触发计划。

注意，触发计划不能用来代替事件响应文档，例如应急手册。风险模型不是在响应事件中必须使用的一个参考工具，相反，它（包括触发计划）应该是用来决定风险发生后续行为的一个管理工具。

使用风险模型来制订计划

当你创建风险模型后，所有的计划会议都应该参考它。这不仅包括产品管理的长期计划会议，也包括与工程师一起进行的敏捷开发计划会议。

在每个计划会议中，应该检查最重要的一些风险[注4]。会上应当询问的问题列表如下所示：

- 与上次检查相比，这个风险现在是不是更严重了？
- 我们是否应该在这次计划排期中，安排人员来解决某些风险？
- 我们是否应该在这次计划排期中，安排人员来执行缓和风险的措施，降低它们的可能性或者严重性。

每个计划会议都应该包括对风险模型的检查，并且风险模型中的每一项（不管是解决风险还是缓和风险）都应该与其他工作一起排列优先级。

如果你的团队在计划会议中使用 Jira 或者 Pivotal Tracker 这样的工具，你可能希望将最关键的风险添加到这些跟踪工具中。如果这样做的话，你应该在跟踪工具中添加风险 ID 这一项，同时在风险模型中添加一列跟踪 ID，记录这些风险在跟踪工具中的 ID。

维护风险模型

风险模型的最大挑战是模型本身很容易变得过时。我们的天性会让我们建立完模型后，将它扔到某个抽屉中并遗忘它。

如果你不花时间来维护风险模型，那么它很快会变得过时和无用。

为了始终保持风险模型的内容及时准确，你应该与相关权益人（包括开发团队和合伙人）定期对它进行检查。这个时间可以是每月，但不应该超过一个季度。确切的重复周期应该根据你的业务流程来制订。如果你马上要进入一个计划周期，那么在周期开始前进行一次定期的模型检查是最理想的情况。

风险模型检查参与人

注意，你应该定期变换检查风险模型的人员。通过要求不同人员来检查并评价风险模型，你可以获得崭新的视角，也可以避免让定期检查会议变得毫无意义。

在这次检查中，你需要：

发现新风险

系统中有没有增加新的或者近期识别出来的风险？确保它们都被涵盖在风险模型中。[注5]

注4　最关键的风险指的是严重性最高、可能性最高，或者两者都很高的风险。

注5　但是，建议你一旦确信发现了新的风险，就立刻将它添加到风险模型中，而不要等到下一次的风险回顾会议。你可以在回顾会议中更新风险的所有数据，但是一旦发现就应当立即将它标注出来。

删除旧的风险

模型中的某些风险是否不再存在——因为它们无法再现，或者引发的原因已经被修复？如果是，从模型中删除这些风险项。

更新可能性和严重性

发现可能性和严重性发生变化的风险项。通常，近期实施的风险缓和计划有利于降低风险的可能性或者严重性，或者收集到其他影响可能性或者严重性的信息。如果有，请更新这些风险项。

检查优先级高的风险

检查所有高可能性或者高严重性的风险。分别对它们进行讨论，确保所有信息都准确无误。是否可以制订新的缓和计划，或者有任何缓和计划需要更新？关于触发计划呢？你是否对风险进行了监控？如果没有，为什么不监控？你还能做哪些事情来改善风险情况？

检查优先级低的风险

如果时间允许，继续检查可能性和优先级低的风险。你不用每次都检查每项风险，但是需要确保经常查看高优先级的风险。除此之外，你还可以单独安排一个会议来详细检查所有低优先级的风险，确保它们没有被人遗忘，以及确保不会隐藏或错过任何应该提升它们优先级的机会和原因。

与管理团队共享你的风险模型

你应当与产品管理和上级管理团队分享你的风险模型。这对于那些无法每天直接与之交流的团队来说，是一个有效的沟通工具，能够帮助大家在某些特定问题上达成一致的认识。

最近我发现在开管理大会之前，有一个很不错的做法。指定某人收集全公司的所有风险模型，将它们整合到一张大表中。然后，只保留高可能性或者高严重性的风险，其余的都删除掉。在整个管理大会上，根据这个核心的“高 / 高”风险列表来讨论公司所有产品的风险，统一不同团队对风险模型的理解，并积累最佳实践经验。

49

第 8 章

风险缓和

风险模型中的风险缓和计划列用来说明可以进行或者正在进行哪些风险缓和措施，来降低当前风险的严重性、可能性，或者二者皆有。我们要做的就是把它从一个高 / 高[注1] 风险降低到一个高 / 中风险或者中 / 高风险。[注2] 我们不是要完全消除风险，只是降低风险发生的可能性和严重性。

正如第 7 章的“缓和计划”一节中所述，你可以按照一个基本流程来缓和风险。缓和计划详细描述了你将要（立即或者即将）为了降低风险可能性或严重性做的事情。

风险缓和是为了知道当问题发生时应该做什么事情，能够尽可能降低问题所带来的影响。缓和措施是为了确保即使在服务或者资源发生故障的时候，系统也能够尽可能完好地工作。

我们先以一个缓和计划为例进行说明。假设有一个系统中使用的数据库，如第 5 章中所述。我们再进一步假设，这个数据库运行在一个高质量的硬件上，并提供了 RAID 磁盘阵列或者服务器级别冗余硬件的备份功能。我们相信这个数据库是高度可靠并且高度可用的。因此，在我们的风险模型上，我们将数据库失败标记为低可能性。

50 但是，数据库仍然存在单点故障的可能。如果这个数据库服务出现问题（虽然可能性比较小），你的整个系统都会受到影响。因此，在我们的风险模型上，它的严重性是高的。

于是，这个风险是一个低 / 高风险，非常类似于第 6 章中“订单数据库：低可能性，高严重性”一节中所描述的场景。

我们如何能缓和这个风险呢？首先，一个办法是添加多个读副本数据库，以图 8-1 所示

注 1　关于“高 / 高”的定义，请参考第 6 章对严重性和可能性的介绍。
注 2　或者更低的其他组合，例如从中 / 高变成中 / 中、中 / 低或者低 / 低。

的热备的形式运行。如果主数据库出现故障，热备数据库可以极大降低系统停止服务的总体时间。该办法可以降低风险的严重性，甚至可能将它降低为一个低 / 中级风险。

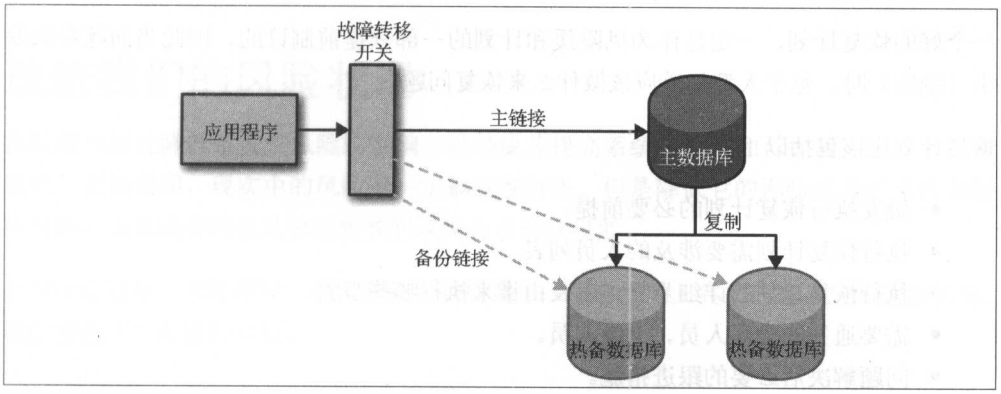


图8-1：用于缓和风险的热备份数据库示例

这就是一个缓和计划。

那么风险缓和和风险管理之间的区别是什么呢？它们有着类似却不同的概念。

风险缓和

风险缓和指的是通过降低风险发生的可能性，或者降低风险发生时的严重性，来降低风险的影响。

风险管理

风险管理指的是在解决风险和缓和风险之间做出选择。我们需要考虑是否能够经济、及时地解决风险，还是仅仅只需降低风险的影响即可。

恢复计划

51

如果已知的风险真的发生了，你必须处理它的结果。你可以通过一个恢复计划建立一系列操作，处理和修复风险所造成的影响和问题。恢复计划通常不会影响可能性，只会影响风险的严重性。

恢复计划是风险缓和的一种特殊类型，专门用来降低风险发生的严重性。恢复计划描述了一个已知风险发生后你应该采取的行为，包括以下几种：

- 尽可能迅速地停止问题的行为。
- 降低问题影响的临时行为。

- 通知用户问题情况的消息内容，以及他们可以采取的降低影响的行为。
- 上报问题的流程，以及需要通知的公司内部人员。这能够让公司统一了解问题情况，一起处理问题和任何可能带来的波及影响。

一个好的恢复计划，一定是作为风险缓和计划的一部分提前制订的，因此当问题真的发生（触发）时，每个人都知道应该做什么来恢复问题。

恢复计划应该包括以下内容：

- 触发执行恢复计划的必要前提。
- 执行恢复计划需要涉及的人员列表。
- 执行恢复计划的详细步骤，以及由谁来执行哪些步骤。
- 需要通知的管理人员、上级人员。
- 问题解决后必要的跟进措施。

恢复计划应该保存在团队都知晓的地方，即当危机发生时每个人都知道在哪里找到它。这个地方可以是支持手册或者内部支持网络上。当执行某个恢复计划之后，应该对故障进行事后分析，同时也应该对恢复计划进行分析，以决定是否存在任何改进的措施，或者确保所有改动都经过批准。

如果针对于某个具体风险的恢复计划是有效的，那么就说明这个风险缓和计划也是有效的，可以用来降低指定风险的严重性。

52

恢复计划

对于一个灾难性的数据库故障来说，图 8-1 所示的复制过程就是一个恢复计划的开始。但是，要形成一个完整的恢复计划，还需要引入一个实施故障转移的过程，确定何时进行故障转移的标准，执行故障转移的确认流程，以及在故障转移后的清理工作。

容灾计划

容灾计划是一种恢复计划，主要用来描述当某种灾难发生时公司应该采取哪些措施。这类灾难通常严重性很高，但是可能性很低。

一个符合容灾计划的灾难例子就是，一个或多个数据中心不可用（不管这是由于技术问题、自然灾难还是严重安全漏洞导致的）。

你可以像建立恢复计划那样建立和管理容灾计划。二者之间唯一的区别在于要缓和的风

险的严重性、细节程度以及执行计划的投入。

一般来说，容灾恢复计划在公司和管理团队内部都会更加重视。针对这些灾难类型，可能需要提前安排具体业务的恢复时间。但是，这些都不是与恢复计划显著区分的地方。

改进我们的风险状况

风险缓和通过降低系统中风险的影响，已经成为提高系统可靠性和可扩展性的一个重要途径。它也说明，现实中的风险不一定都能够解决，但是降低它的影响或者严重性却很有可能，而且通常这也足够达到我们期望的可用性程度。

在与风险模型一起使用时，风险缓和计划提供了一个有用的工具来改进系统的健康状况。我们会在下一章进行介绍。

53

第 9 章

比赛日

我们经常容易犯一个危险的习惯错误，就是建立恢复计划和容灾计划后，将它们搁置在抽屉里，只有在需要的时候才会想起它们来。

如果你真的这样做，几乎可以保证，当你需要恢复 / 容灾计划的时候，它们已经过时了。除此之外，如果你不保持更新它们，就可能引入大量其他的问题，导致计划无法执行，或者在实际中无法成功执行。

因此，你应该定期对恢复 / 容灾计划进行测试。应该把定期测试这些计划和其他风险缓和措施，作为你的公司文化的一部分。

我们用来测试这些计划的方法被称为举行“比赛日”。“比赛日”指的是通过测试来触发系统中某个失败模型，然后观察你的操作人员和工程师如何进行响应，包括他们如何执行恢复计划和容灾计划。在比赛日过后，团队通过事后复盘来发现计划中的问题和后续改进。这些改进会不断更新恢复 / 容灾计划，当问题真正发生时可以派上用场。

预发布环境和生产环境

你可能想知道，是否应当在预发布环境或者生产环境中来测试恢复计划？这个问题很难回答，也没有标准答案。我们先分析一下以下这些选项。

预发布 / 测试环境

在预发布 / 测试环境中测试恢复计划是最安全的。预发布或者测试环境允许你执行一些无法在生产环境中执行的破坏性测试。另外，你不用害怕这些测试可能造成的失误。如果你决定通过预发布 / 测试环境来测试恢复计划，请记住以下内容。

54

- 确保预发布 / 测试环境与生产环境是完全隔离的。这个环境不应该依赖于任何生

产资源，生产资源也不应该依赖任何测试环境的资源。请参考图 9-1。

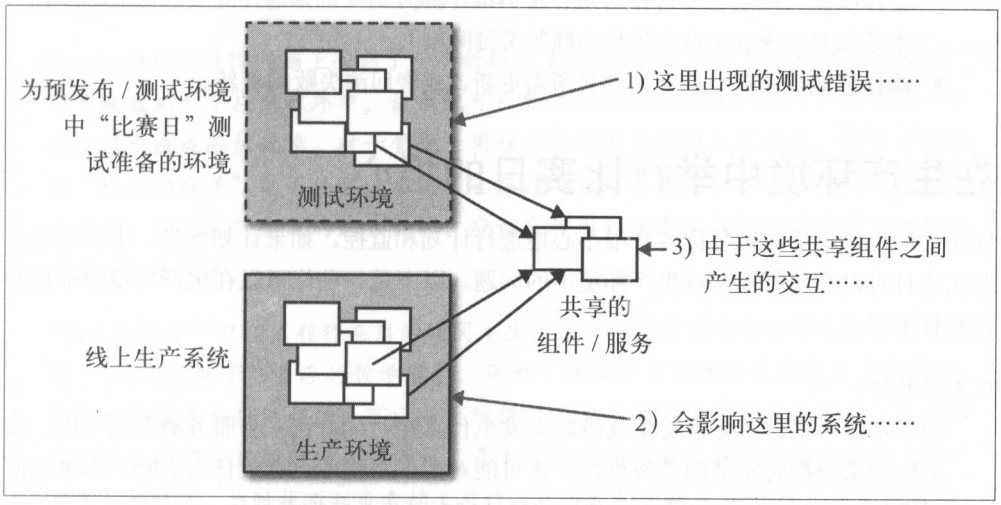


图9-1：不隔离测试环境的危险性

- 确保预发布环境和测试环境与生产环境尽可能保持一致。虽然使用预发布 / 测试环境来测试恢复计划是可行的，并且你也可以使用这类环境来测试各种破坏性的故障场景。但是，它们并不能保证在生产环境中一定得到相同的结果。这是因为生产环境的服务器规模更大，承载着更大的数据集，并且管理着远超其他环境的实时流量。这些差异使得某些测试在非生产环境下变得无效。理想情况下，测试环境应该达到跟生产环境一样的规模，并且使用与生产环境一样的数据，但是，这通常在成本和维护方面无法实现。如果你认为某个测试必须在与生产环境规模一样的情况下进行测试，那么也许应该考虑在生产环境中进行测试。

生产环境

55

在生产环境下测试风险和恢复计划看上去没有道理。为什么要让生产环境强制出现一次故障，只是为了确定它不会发生故障？答案很简单：如果你在团队就绪并且准备充分的情况下（换句话说，不是在午夜），同时在一个对用户影响最小的时间点上，在仔细考虑测试的每个步骤后，可以安全地在生产环境下进行实际测试，从而了解到恢复真正故障所要付出的努力。

如果你决定在生产环境中测试你的恢复计划，请记住以下内容：

- 小心对当前正在使用系统的用户造成影响。
- 从业务角度考虑测试。是选择给用户增加风险，还是选择根据这些测试来降低长期风险，这其中需要权衡。

- 选择你的团队成员状态最好的时刻（通常是工作日白天大家都在办公的时候）来进行测试，但是也要选择对用户影响最小的时刻（例如选择流量较低的时候，而不是在月底或者季度末销售高峰等关键时刻）。
- 确保你能够快速、简便地部署所需更新，或者回滚失败的更新。

在生产环境中举行比赛日的担心

你应当对生产环境中进行比赛日很小心地进行计划和监控。如果计划恰当，生产环境中的比赛日可以非常好地暴露生产环境中的问题。以下是一些你可以在生产环境中举行的比赛日示例。

服务器故障

如果系统中某台服务器发生故障时会发生什么呢？试着让一台服务器无法工作。如果你的系统拥有充足的备份机器，这可能对生产系统不会造成任何影响。你可以使用排除法来发现这类问题，并利用恢复计划来替换故障服务器。

网络分区

如果发生网络故障或者网络分区会产生什么影响呢？如果经过仔细计划，你可以在
56 避免对生产环境系统造成重大影响的情况下，模拟网络分区的情况。它们可以用来测试系统的通知和跟进行为，以及你的团队对事件的响应能力。

数据中心故障

如果整个数据中心发生故障会发生什么呢？如果经过仔细计划，你的系统应该能够处理这类事件。你会如何响应这类故障呢？

随机故障

如果系统中存在更小的随机错误会发生什么呢？你的应用程序是否能从这些错误中适当地进行恢复？

从很多方面看，最后一项都是最令人害怕的。为什么？因为你可以想象出来当服务器或者数据中心发生故障时会发生什么。你可能已经制订了处理这些情况的计划（如果你还没有，应该立即动手制订）。但是对于一个“随机”问题来说，即使范围很小，也像是一些你无法掌控的东西。说实话，它的确是。但是，正是这些随机事件，成为你在构建高可用、低风险系统时最大的绊脚石。

捣乱的猴子

Netflix 将随机故障问题上升到了一个新的高度。他们开发了一个称为“捣乱的猴子”的系统放到整个应用程序中。该系统可以随机或定期地在生产环境中以及用户使用过程中造成随机故障。对于管理应用程序工程师和运维人员来说，他们并不知道“捣乱的猴子”做了什么。相反，它假设工程师们已经准备了恰当的恢复和缓和流程，因此它所造成的这些故障，应该在对用户没有造成影响的前提下被解决或者临时处理掉。

“捣乱的猴子”只在工程师在岗的时间发生，这样他们可以响应任何无法自愈的问题。“捣乱的猴子”背后的哲学理念，是为了鼓励并且实际要求去构建高可用的、可自愈的服务和应用程序，可以在无须任何人工干预的情况下自我恢复。它选择在白天工程师在岗的时候进行测试，是为了避免问题发生在系统更加繁忙（用户更多）的晚上，临时找不到负责的工程师。这个新的尝试在 Netflix 已经取得成功。

“捣乱的猴子”是比赛日测试的一个最佳实践范例，并且 Netflix 也对其比赛日的架构完成了一些非常令人称赞的事情。但是，它花费了巨大的付出，巨大的资源，以及在 Netflix 能够满足安全、有效在生产环境中运行“捣乱的猴子”之前，巨大的代码修改工作。因此，“捣乱的猴子”不应该是你进行生产环境比赛日测试的第一步。但是，如果你的公司有足够的能力和付出，可以将它作为一个发展的目标。

57

比赛日测试

比赛日测试，是一个能够帮助生产环境在系统层面保证完全运行的重要手段。它允许你通过一种安全的方式，来验证自己的支持计划和流程在问题发生时，是否能够真正正确无误地运行。

如果一切都完成得很好，比赛日测试可以极大改善系统的可用性，并降低生产环境中存在严重问题的风险。

59

第 10 章

构建低风险系统

在第 8 章中，我们了解到如何缓和系统中已有的风险。但是，还有一些技巧可以帮助你主动构建低风险的系统。本章会来介绍这些技巧中的一部分。虽然这远不是所有内容，但至少会让你在构建和扩展系统时开始思考如何去降低风险。

冗余

冗余是一个提高系统可用性和可靠性的显著方式，同时也能降低系统的风险状况。但是，冗余会给系统增加复杂性，这又增加了系统的风险。因此，重要的是控制增加冗余的复杂度，以及实际中是否能对风险状况带来可测量的改进效果。

以下是一些“安全”的冗余改进示例：

- 将应用程序设计为可同时安全运行在多个独立硬件上（例如并行服务器或者冗余数据中心）。
- 将应用程序设计为可以独立运行任务。这可以在不过多增加复杂性的前提下，帮助恢复故障资源。
- 将应用程序设计为可以异步运行任务。这可以在避免影响主程序处理的前提下，通过队列来延迟运行任务。

60

- 将本地状态存储到一个指定区域。这可以降低系统其余部分对状态管理的需要，提高使用冗余组件的能力。
- 尽可能使用幂等接口。幂等接口指的是可以重复调用的接口，无须担心一个动作被执行多次所带来的影响。幂等接口通过简单的重试机制可以实现错误恢复。

幂等接口示例

我们以一辆具有控制行驶速度接口的汽车为例，通过以下示例来说明幂等接口：

- 将我当前的速度设置为 35 英里 / 小时。

而下面是一个非幂等的接口：

- 将我当前的速度增加 5 英里 / 小时。

幂等接口可以被多次调用，但是只有第一次调用会起作用。连续或者重复的调用对改变汽车速度没有任何作用。重复让汽车的行驶速度为 35 英里 / 小时，与调用一次的效果是一样的。但是，非幂等接口在每次调用时都会对汽车的速度产生影响。重复让汽车增加 5 英里 / 小时会让汽车的行驶速度越来越快。

对于一个幂等接口，汽车的“驾驶员”只需要告诉汽车需要的行驶速度即可。如果出于某种原因，行驶速度为 35 英里 / 小时的请求没有被汽车执行，那么驾驶员只需简单地（并且安全的）重复发送请求，直到它确定汽车接收到了请求，然后驾驶员就可以确定汽车实际上的行驶速度是 35 英里 / 小时。

但是对于非幂等接口来说，如果汽车的“驾驶员”希望按照 35 英里 / 小时的速度行驶，它需要发送一系列命令让汽车加速，直到它按照 35 英里 / 小时的速度行驶。如果其中一个命令失败，驾驶员需要通过某种机制知道汽车的当前速度，并决定是否重新发送“加速”命令。它不能只是简单地重发加速命令——必须首先弄清楚是否需要发送这个命令。这显然是一个更加复杂，也更加容易出错的操作。

使用幂等接口相比起使用非幂等接口来说，会让驾驶员执行更简单的操作，降低出现错误的几率。

增加了复杂性的冗余改进

61

哪些例子属于增加了复杂性的冗余改进呢？实际上，至少对大多数应用程序而言，很多冗余改进看上去似乎有用，但是却增加了复杂性，实际效果弊大于利。

假设我们要创建一个系统的并行版本。在这个版本中，如果其中一个系统出现故障，另一个系统会替代它实现所需的功能。虽然这对于那种要求极度高可用的系统（例如航天飞机）非常重要，但是它通常过于复杂，导致复杂性急剧增加，而复杂性增加的同时意味着风险增加。

另一个例子是明显分开的活动。微服务是一个能够显著提高系统质量、降低风险的模型。

第 12 章包含了关于服务和微服务的更多内容。但是，如果极端来说，将系统划分成过于细粒度的微服务可能会导致增加系统的整体复杂性，同时增加了风险。

独立性

共享某个组件的一些组件可能会声称它们之间是独立的，但是实际上，它们都依赖于那个通用组件，如图 10-1 所示。

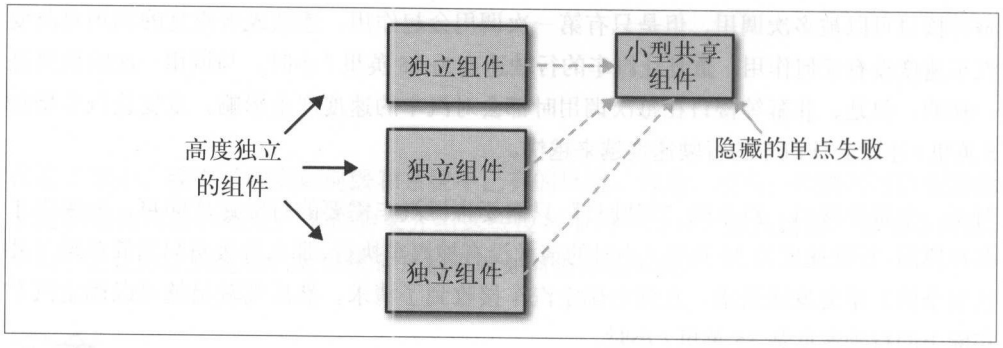


图10-1：依赖于共享组件会降低组件之间的独立性

如果这些共享组件很不引人注意，或者根本没人知道它们的存在，那它们就可能成为系统中的故障单点。

假设有一个应用程序运行在 5 台独立的服务器上。你可以通过这 5 台服务器来增加系统整体的可用性，降低单台服务器故障导致的系统不可用。图 10-2 展示了这个应用程序。

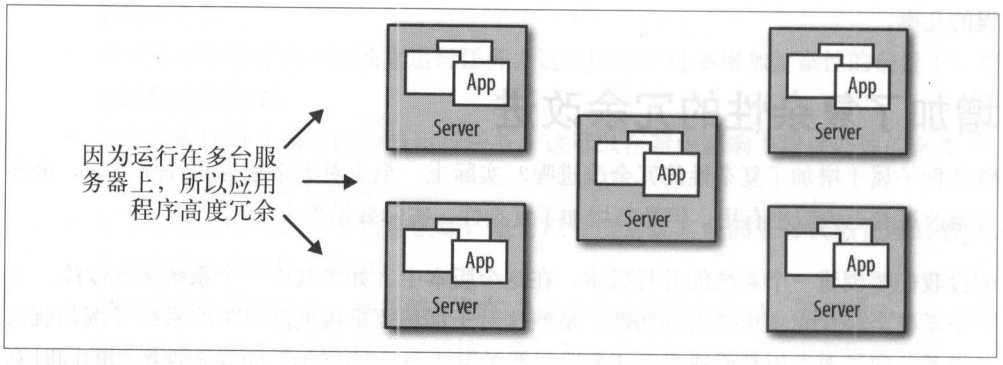


图10-2：独立服务器……

62 但是如果这 5 台服务器实际上是运行在同一台硬件服务器上的 5 个虚拟机呢？或者这些

服务器都运行在同一个机柜上？如果这个机柜的电源发生故障会怎样呢？如果共享的硬件服务器出现故障又会怎样呢？示意图如图 10-3 所示。

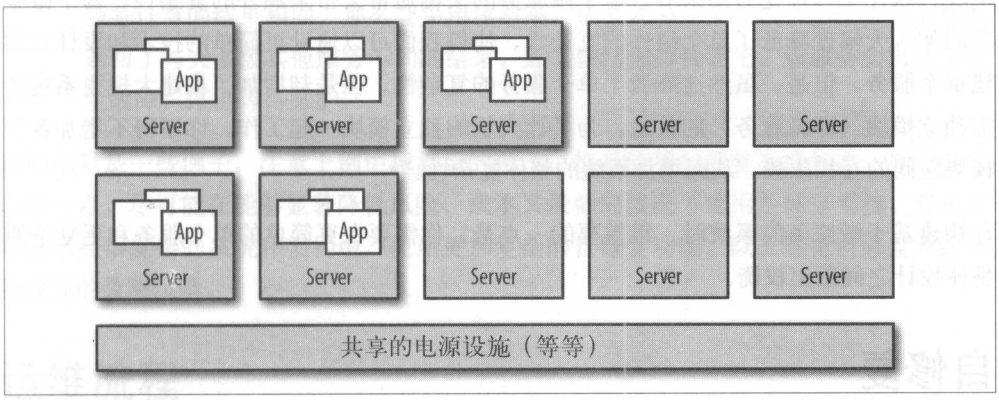


图10-3: ……不是像你想象的那么独立

因此，你的“独立服务器”可能并不像你想象的那么独立。

安全

在软件系统中，坏人永远是个问题。即使在出现大规模 Web 应用程序之前，安全和安全监控也永远是构建系统过程中需要考虑的一部分。

但是相比之前，Web 应用程序已经变得更大、更复杂，存储着更多的数据，处理着更大的流量。随着这些系统中的数据越来越有价值，试图危害系统的坏人也急剧增加。他们或者试图获取高度敏感的数据，或者让大型系统无法使用。有些人出于经济利益，有些人只是一时兴起。不管动机如何，结果如何，坏人都正在成为一个越来越大的问题。

63

Web 应用程序的安全问题超出了本书的内容范围[注1]。但是，无论是对于高可用性，还是对于降低大规模系统的风险来说，高质量的安全措施都是必不可少的。这里的关键点在于，你应该在风险分析、风险缓和以及开发过程中均考虑到安全因素。但是，其中应该包含的内容超出了本书的范围。

简单性

复杂是稳定的敌人。系统越复杂，就越不稳定。越不稳定，风险性就越大，从而降低可用性。

注 1　O’Reilly 已经出版了多本关于安全和 Web 安全的书籍。

虽然我们的应用程序逐渐变得越来越大，并且明显愈加复杂，但是如何在设计架构和实现时始终保持简单性，对于保证应用程序可维护、安全和低风险至关重要。

从现代软件架构理论方面来看，基于微服务的架构通常会带来额外的复杂性。基于微服务的架构大幅度降低了单个组件的复杂性，使得我们可以通过更简单的技术和设计来创建单个服务。但是，虽然这降低了单个服务的复杂性，但是却增加了构建大规模系统所需独立模块（即微服务）的数量。为了让大量的独立模块一起工作，你不得不增加各个模块之间的互相依赖，从而增加系统的整体复杂性。

在构建基于微服务的系统时，很重要的一点是，你需要在更简单的单个服务和更复杂的整体设计之间做出权衡。

64 自修复

在系统中设计自纠正和自修复的流程，可以降低可用性故障的风险。

正如我们在第 3 章所讨论的，如果你希望达到 5 个 9 的可用性，每个月的宕机时间不能超过 26 秒。即使你只达到 3 个 9 的可用性，每个月也只能容忍 43 分钟的宕机时间。如果某个服务出现故障后，需要某人在半夜起来发现、诊断并修复问题，那么这 43 分钟显然很难够用。一次事故就可能让你无法达到每月 3 个 9 的目标，更别说要达到 4 个 9 或者 5 个 9 的目标，你必须能够在没有任何人工干预的情况下自动修复问题。

这就是为什么要建立自修复系统。自修复系统听上去好像是一种高级、复杂的系统，但实际上并不一定如此。一个自修复系统可能只是部署在多个服务器之前的负载均衡器，当某个服务器无法处理请求时，可以将请求快速路由给其他服务器。这就是一个自修复系统。

自修复系统有很多级别，从简单到复杂。下面举例进行说明。

- 一个负载均衡器，能够将当前服务器无法处理的请求，重新路由到新的服务器。
- 一个“热备份”数据库，与主生产数据库保持同步。如果主生产数据库出于任何原因出现故障或者下线，热备份数据库会自动承担起主数据库的角色，并开始处理请求。
- 一个重试失败请求的服务，这样即使原有请求临时遇到问题，新的请求也能够成功。
- 一个保留待处理工作的队列系统，如果某个请求失败，它可以稍后被重新调度到一个新的工作进程上，从而增加了完成工作的可能性，避免了丢失工作记录的可能性。

- 一个时刻运行并试图导致系统故障的后台进程（例如，像 Netflix 中“捣蛋的猴子”一样的系统），可用来检查并确保系统可以自己恢复到正常状态。
- 一个向多个独立开发和管理的服务发送请求，但进行相同计算的服务。如果所有服务计算的结果相同，那么会采用该结果。如果其中一个（或者多个）独立服务返回了与大多数其他服务不同的结果，那么该结果会被弃用，该服务会被认为发生故障，停机等待维护。

这些都只是一些例子。注意上例中越靠后的例子，给系统增加的复杂度越高。你需要小心这一点，尽可能在能够显著降低风险、成本又最低的情况下使用自修复系统，避免在本来已经很复杂的系统和架构中，使用更加复杂的自修复功能，以及避免自修复系统本身存在的故障风险。 65

运维流程

软件系统会涉及人，是人就会犯错。通过可靠的运维流程，你可以将系统中人的影响降低到最小，并且减少人参与操作的过程，降低错误发生的可能性。

文档化、可重复的流程可以降低人工操作中的一个重要的问题——健忘：忘记执行步骤、执行了错误的顺序，或者执行某个步骤时出现失误。

但是文档化、可重复的流程也只能降低这个问题的发生率。人还存在其他的问题。人会犯错，他们会按错键盘、他们以为自己知道在干什么但实际却不知道。他们的每一次操作都不太一样，每一次操作也无法审查，甚至可能在情绪不好的时候执行错误的操作。

如果你能够将系统中需要人执行的操作自动化，那么就能降低失误发生的几率，提高任务完成的可能性。

重启服务器

假设你出于某个特殊目的（我们不去管这是不是一个好主意），需要定期重启某台服务器（或者一系列服务器）。

你可能会让用户登录服务器，成为超级用户，然后执行“reboot”命令。但是，这会导致几个问题：

- 现在只要有任何人需要执行该命令，你都需要授予他们登录生产服务器的权限。此外，他们还必须有超级用户的权限才能执行重启命令。
- 当某人以超级用户身份登录服务器后，他可能不小心执行了其他命令，导致服

务器出现故障。

- 当某人以超级用户身份登录服务器后，他们可能会恶意执行一些危害服务器的操作，例如在 Linux 上运行 rm -rf / 命令。
- 你可能没有任何关于操作发生的记录，也没有谁执行重启、为何重启的记录。

相对于通过人工操作来重启服务器，你可以实现一个自动执行重启的流程。除了能够执行重启之外，它还可以提供以下好处：

- 减少对生产服务器身份授权的发放，消除失误和恶意行为出现的可能性。
- 可以记录所有执行重启的操作。
- 可以记录谁发起了重启操作。
- 可以验证发起重启操作的人是否有相应的权限（细粒度权限——你可以将重启服务器的权限授予一个用户组，而不给他们任何其他的访问权限）。
- 可以确保在服务器重启前执行其他任何必要的操作。例如，临时将服务器从负载均衡器上删除，或者优雅地停止运行中的应用程序等。

你可以看到，通过自动化整个流程，你可以避免人为失误，并能够更好地控制由谁来如何执行操作。

第III部分

服务和微服务

服务是一个为构建某个或更多大型产品所提供的功能性、有明确边界的系统。

69

第 11 章

为什么使用服务

通常来说，应用程序都是一个独立、大型、明确的单体程序。这个独立的单体程序包含了一个应用程序的所有业务逻辑。为了改进某一项业务功能，一个开发人员必须在这个应用程序的内部进行修改，同时，其他所有开发人员也必须在其中进行修改。这样，开发人员之间很容易造成干扰，并且互相冲突的修改会导致更多的问题和故障。

在一个面向服务的架构中，单个服务只包含所有业务逻辑的某个子集。这些独立的服务互相连接，为应用程序提供完整的业务逻辑。

单体应用程序

图 11-1 展示了一个大型的、单一实体的应用程序，它的架构很复杂，让人难以理解。

这也是大多数发展成单体应用程序的系统看上去的样子。在图 11-1 中，你可以看到 5 个独立的开发团队在相互重叠的部分同时工作。我们无法知道在某一个时间点，究竟是谁在哪一块上工作，也不难想象代码变更所导致的冲突和问题。代码质量，以及因此产生的系统质量和可用性也会受到影响。此外，系统变得越来越复杂，单个开发团队想要更改代码也变得越来越困难，不得不面对与其他团队的协作、互相冲突的改动，以及组织内部纠缠不清等问题。

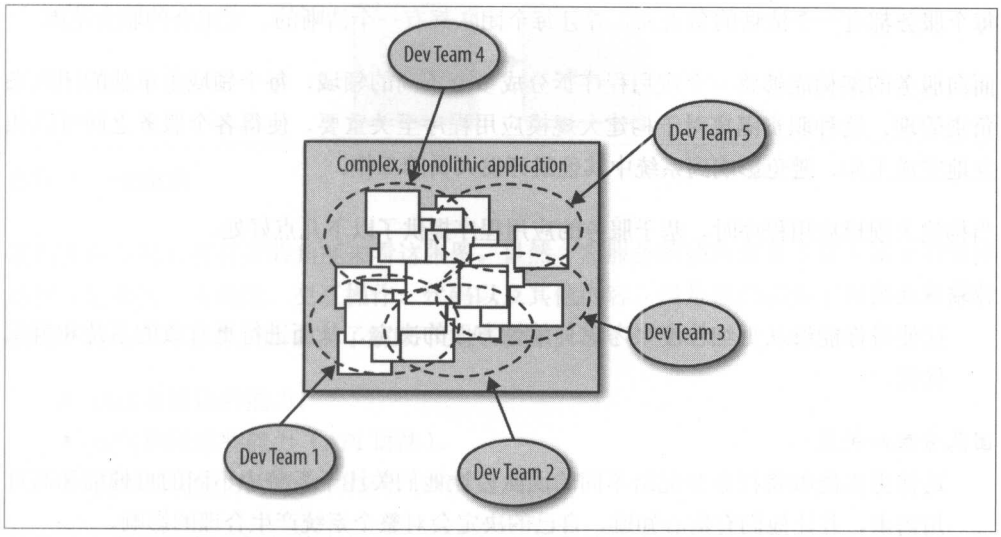


图11-1：一个大型的、复杂的单体应用程序

基于服务的应用程序

图 11-2 展示了由一系列服务所组成的同样的应用程序。

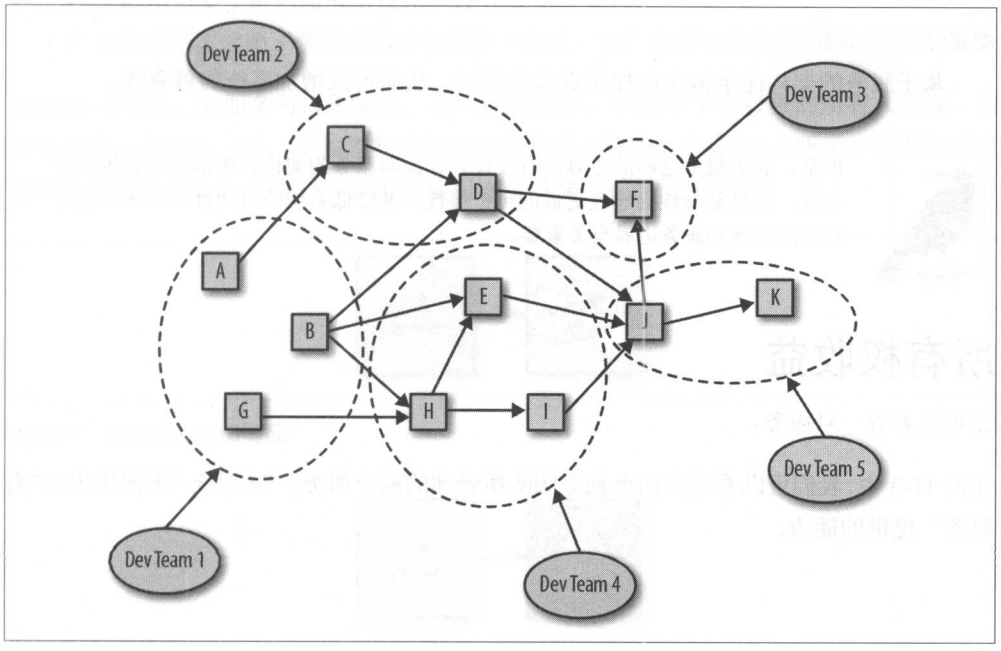


图11-2：一个大型的、复杂但基于服务的应用程序

71 每个服务都有一个清晰的负责人，并且每个团队都有一个清晰的、无重合的职责范围。

面向服务的架构能够将一个应用程序拆分成多个不同的领域，每个领域由单独的团队来负责管理。这种职责隔离对于构建大规模应用程序至关重要，使得各个服务之间可以独立地完成工作，避免影响到系统中其他组开发人员的工作。

当构建大规模应用程序时，基于服务的应用程序提供了以下几点好处。

伸缩性决定

这使得你能够从更细粒度来考虑伸缩性方面的决定，从而进行更有效的系统和组织优化。

团队分配和关注

这使得你能够将任务分配给不同的团队，让他们关注于系统中不同的可伸缩和高可用需求，并让他们有信心知道，自己的决定会对整个系统产生合理的影响。

复杂的本地化

使用基于服务的架构，你可以将各个服务看作一个个黑盒，只有服务的所有者需要了解该服务内部的复杂逻辑。其他开发人员只需要知道服务所能提供的能力，而不需要知道它内部的工作原理。这种认知和复杂性上的隔离，能帮助你创建更大型的应用程序，并且更有效地管理它们。

测试

基于服务的架构比单体应用程序更容易测试，从而可以增加系统的可靠性。

但是，如果服务之间的边界没有设计好，面向服务的架构会增加系统整体的复杂性。这种复杂性会导致更低的可扩展性，并降低系统的可用性。因此，选择合适的服务和服务边界至关重要。

72 所有权收益

让我们来看一对服务。

在图 11-3 中，我们可以看到由两个独立团队所管理的两个服务。“左服务”正在使用由“右服务”提供的能力。

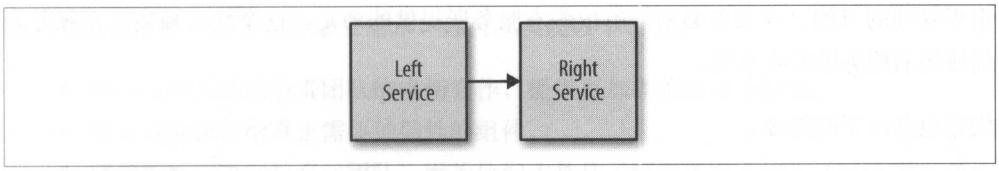


图11-3：一对服务

我们先从左服务所有者的角度来看这张图。显然，左服务的团队需要了解左服务的整体结构、复杂性、正确性、交互操作、代码以及其他内容。但是他们需要了解右服务的什么呢？首先，左服务团队需要了解以下几件事情：

- 该服务提供的能力。
- 如何调用那些能力（API 语法）。
- 调用那些能力的意义和结果（API 语义）。

这些是左服务需要了解的基本信息。那它们不需要了解右服务的哪些信息呢？这包含很多内容，例如：

- 它们不需要了解右服务是一个单独的服务，还是多个子服务的组合。
- 它们不需要了解右服务需要依赖其他什么服务。
- 它们不需要了解右服务是用什么语言编写的。
- 它们不需要了解右服务所使用的硬件或系统架构。
- 它们甚至不需要了解谁来提供右服务（但是，它们需要知道遇到问题时联系谁）。

如图 11-4 所示，右服务可以很复杂，也可以很简单。但是对于左服务的所有人来说，右服务就像一个黑盒一样，不知道其内部的结构是什么样的，如图 11-5 所示。只要知道这个黑盒的接口是什么（API），就可以使用这个黑盒所提供的能力。

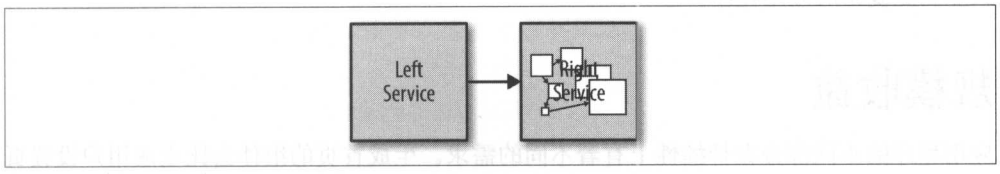


图11-4：右服务的内部情况

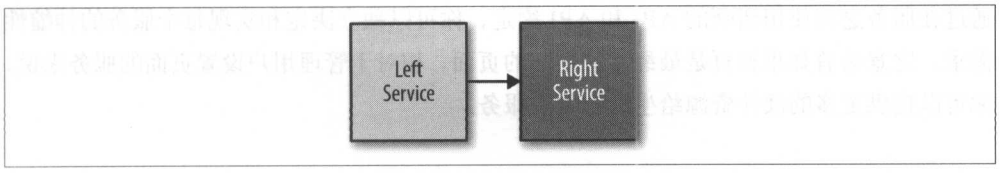


图11-5：右服务的复杂性被隐藏起来

73 出于管理的目的，左服务必须能够依赖右服务所提供的约定。这个约定描述了左服务如何使用右服务的所有事项。

约定包含以下两部分。

服务的能力（API）

- 服务的用途。
- 如何调用服务以及每个调用的意义。

服务的响应性

- API 使用的频率？
- 什么时候可以使用 API？
- API 响应的速度？
- API 是否存在依赖？

所有这些信息组成了右服务提供给左服务的约定。只要右服务遵守这个约定，左服务就不需要知道或者关心右服务是如何来实现这些约定的。

约定最后一段关于响应性的部分，被称为服务等级协议，或者 SLA。它是让左服务能够放心使用右服务，而不用去关心右服务如何实现的关键指标。

我们将在第 18 章重点讨论 SLA。

74 通过让每个服务都有清晰的所有权，团队可以只关注于他们所负责的模块，以及依赖服务所提供的 API 约定。这种职责隔离使得组织能够更容易容纳更多的团队，因为团队之间的耦合程度更加松散，所以团队之间的距离（不管是组织上的层级还是物理上的距离）也就无关紧要。只要双方之间保持着一致的约定，你就可以扩展组织的规模，创建出更大型、更复杂的应用程序。

规模收益

应用程序的不同部分在伸缩性上有着不同的需求。生成首页的组件会比生成用户设置页面的组件，会更频繁地被使用到。

通过在服务之间使用清晰的 API 和 API 约定，你可以独立决定和实现每个服务的伸缩性需求。这意味着如果首页是最经常被访问的页面，相对于管理用户设置页面的服务来说，你可以提供更多的硬件资源给生成首页的服务。

通过独立管理每个服务的伸缩性需求，你可以实现以下事情：

- 通过与相关功能负责团队的密切合作，提供更加准确的可伸缩性。
- 将系统资源留给真正需要伸缩性的组件。
- 将伸缩性的决定权交回团队，因为他们才是最了解服务需求的人（服务所有者）。

75

第 12 章

使用微服务

一个服务提供了应用中其他服务所需要的某些能力。例如，账单服务（提供了用户开具账单功能的组件），账户创建服务（管理创建账户的组件），以及通知服务（包含了通知用户事件和使用情况的功能）。

每个服务都是一个独立的组件。这里的“独立”很重要，独立的服务必须满足以下几个条件。

维护自己的代码库

服务拥有自己的代码库，且区别于其他服务的代码库。

管理自己的数据

服务需要管理其自身数据的状态。服务之间唯一能访问到数据的方式就是通过定义的 API，其他服务不能够直接接触当前服务的数据或者状态信息。

向其他服务提供能力

服务有一个定义良好的能力集合，并将这些能力提供给系统中的其他服务。换句话说，它提供了一个 API。

消费其他服务的能力

服务按照标准的、约定的方式，使用其他服务提供的能力。换句话说，它使用其他服务的 API。

单一所有者

每个服务只能由一个开发团队负责和维护。虽然一个团队可以负责和维护多个服务，但是一个服务只能由一个团队负责和维护。

如何定义服务

你如何来决定，应该将系统中的哪些部分独立出来，形成自己的服务呢？

这是一个好问题，但没有唯一的答案。某些公司喜欢“服务化”，因此将系统拆分成很多（成百上千个）非常小的微服务。另一些公司则将系统拆分成一些更大的服务。这个问题并没有谁对谁错。但是，整个行业在向着更小的微服务的趋势发展。像 Docker 等技术使得这些大量的微服务成为一种可能的系统拓扑。

我们在本书中将交替使用服务和微服务两个名词。

深入了解服务

那么你如何来决定服务的边界呢？公司组织、文化以及应用程序的类型都会在很大程度上影响服务边界的确定。

以下是一组你可以用来确定服务边界的指导原则。不过，这些只是原则，并不是规则，也很可能随着工业进程而发生改变，但是它们可以帮助我们开始思考什么是服务以及如何划分服务。

以下是较高层面上的指导原则（按照先后顺序）。

特定的业务需求
: 是否有特定的业务需求（例如会计、安全或者监管）会影响服务的边界？

清晰和独立的团队所有权
: 负责该功能的团队是否清晰和独立（例如在另一个城市，或者在另一个楼层，或者甚至只是一个不同的经理）是否会帮助确定服务边界？

天然隔离的数据
: 其管理的数据是否天然地与系统其他数据相独立？将数据放到一个单独的数据存储器中是否会对系统造成过大压力？

共享的能力 / 数据
: 它是否提供了一些被其他服务使用的共享能力，是否这些共享能力需要共享数据？

我们现在分别来解释以上每一项原则。

指导原则 1：特定的业务需求

在某些情况下，某些特定的业务需求会决定一个服务的边界。这些需求可能是监管、法律、

安全或者一些关键业务的需求。

示例12-1：支付处理

假设你的系统接受来自用户的在线信用卡支付。但是你应该如何收集、处理并且存储这些信用卡，以及它们所代表的支付呢？

一个好的策略是将处理信用卡的过程交给另外一个服务，与系统中其他的服务区分开。

将处理信用卡这样的关键业务逻辑放在独立的服务中，有以下几点原因：

法律/监管需求

法律和监管需求会要求你在保存信用卡的方式上，选择与其他业务逻辑或其他业务数据不同的方式。将这些拆分成独立的服务，使得我们更容易处理这些数据。

安全

出于安全的考虑，你可能需要额外的防火墙来保护这些服务器。

校验

你可能需要对这些能力进行更加严格的产品测试，来保证它们的安全性。

限制性访问

你通常会希望限制对这些服务器的访问，这样只有必备人员才能访问高度敏感的支付信息，例如信用卡。你通常不希望或者不需要将这些访问权限提供给所有的工程人员。

指导原则2：清晰和独立的团队所有权

应用程序正在变得越来越复杂，需要越来越多的开发人员进行开发，通常也提供更专业的功能。随着开发人员、团队和开发地点的增加，团队之间的协作变得越来越难。

服务能够将更小的、清晰的、独立的模块所有权赋予不同的团队。

78

一般原则

一个单独的服务应该由一个3人到8人的开发团队来负责和运行。这个团队应该负责该服务的所有方面。

这样就减少了团队之间的依赖，使得独立的团队更容易去运行系统和实现创新。

团队所有权

如之前所述，单个服务应该只由单个团队负责和运行，但是单个团队可以负责和运行多个服务。关键是要确定这个团队可以负责该服务的所有方面。这意味着这个团队要负责服务所有的开发、测试、部署、性能以及可用性等各个方面。

但是，根据服务的复杂性和功能不同，一个团队也许有能力管理多个服务。除此之外，如果多个服务本质上是类似的，那么让一个团队来管理所有这些服务可能会更容易一些。

一个团队可以负责或管理多个服务，但是一个服务应该只由一个团队负责和管理。

出于安全原因隔离团队

有些时候，你希望限制能够访问某个服务代码和数据的人员数量和范围。这对于需要审计或者存在法律约束的服务来说尤其重要（如示例 12-1 中所讨论的）。限制对含有敏感数据服务的访问，可以降低数据暴露所导致的问题。像这种情况，你可以从物理上限制只有关键人员才能访问服务的相关代码、数据和系统。

除此之外，将敏感数据分成两个或多个服务，每个服务由不同的团队管理，也可以有效降低多个服务同时泄露数据的风险。

示例12-2：出于安全原因拆分数据

在示例 12-1 中，信用卡数字本身可以被存储在一个服务中。而使用信用卡所必需的信息（例如账单地址和 CCV 代码）可以存储在第二个服务中。通过将该信息分开保存在两个服务中，每个部分都由单独的团队负责，你可以避免任何员工无意或有意地将信用卡信息泄露出去，造成不当影响。

79

你甚至可以选择不由自己来保存信用卡号码，而是将它们存储在一个第三方信用卡服务公司的服务中。这可以保证即使你的其中一个服务被入侵，信用卡本身的数据也不会被泄露。

指导原则 3：天然隔离的数据

服务的一个要求是，其托管状态和数据需要与其他数据相隔离。出于多种原因，让多个独立的代码库对同一组数据进行操作是有问题的。只有在你隔离数据之后，隔离代码和所有权才能起作用。

图 12-1 展示了一个服务（服务 A）试图访问存储在另一个服务（服务 B）中的数据。它说明了服务 A 访问服务 B 中数据的正确方式，即服务 A 通过 API 调用来访问服务 B，

然后让服务 B 访问自己数据库中的数据。

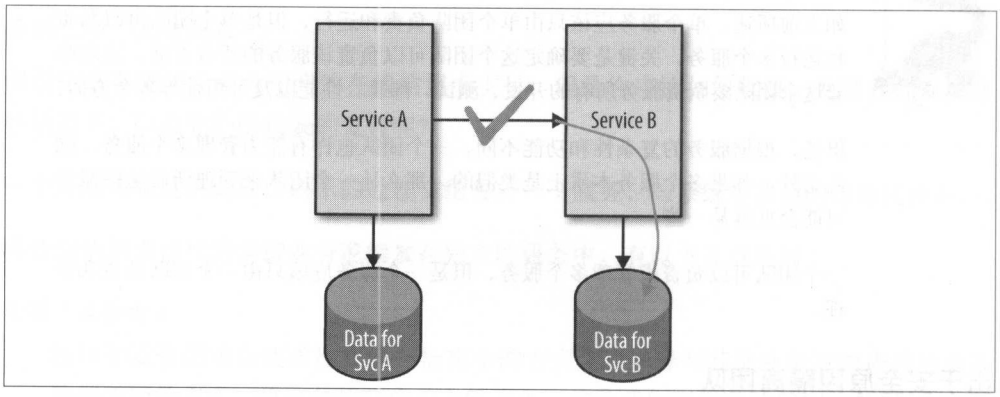


图12-1：共享数据的正确方式

相反，如图 12-2 所示，如果服务 A 试图直接访问服务 B 中的数据，而不是通过服务 B 的 API，那么所有问题都会出现。这种数据集成会导致服务 A 和服务 B 过度耦合，并且当维护数据和迁移模式时会出现问题。一般来说，如果服务 A 越过服务 B 的业务逻辑，直接访问服务 B 的数据，会导致严重的数据版本不一致和数据损坏问题。这种情况应该被严格禁止。

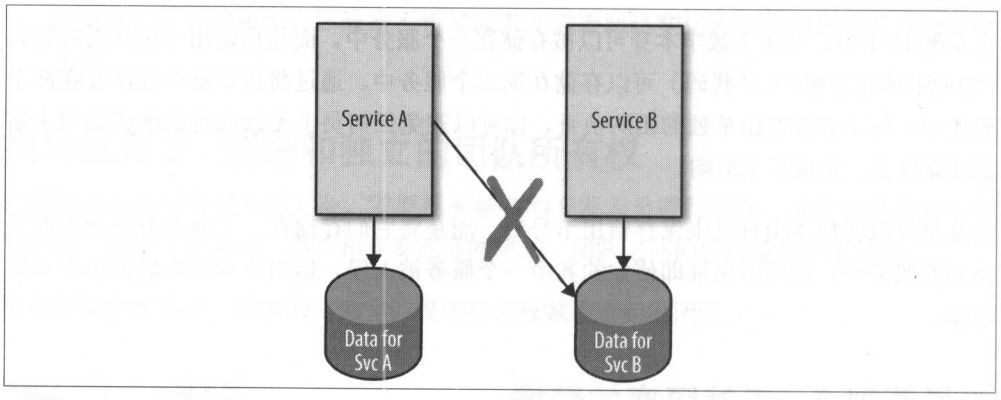


图12-2：共享数据的错误方式

80 如你所见，确定数据划分界线是确定服务划分界线的关键环节。对于一个服务来说，它是否能够负责自己的数据，并通过外部服务接口来提供数据访问呢？如果答案是“是”，说明这是一个很好的服务边界。如果答案是“否”，说明这不是一个很好的服务边界。

一个服务如果需要操作另一个服务的数据，必须通过数据所在服务的公开接口来访问。

指导原则 4：共享的能力 / 数据

有些时候我们可以很容易地创建一个服务，因为它负责提供一系列的能力和数据。这些能力和数据可能需要在很多其他服务之间共享。

说明该原则最好的一个示例就是用户身份服务，它只是简单地提供系统中指定用户的信息。如图 12-3 所示。

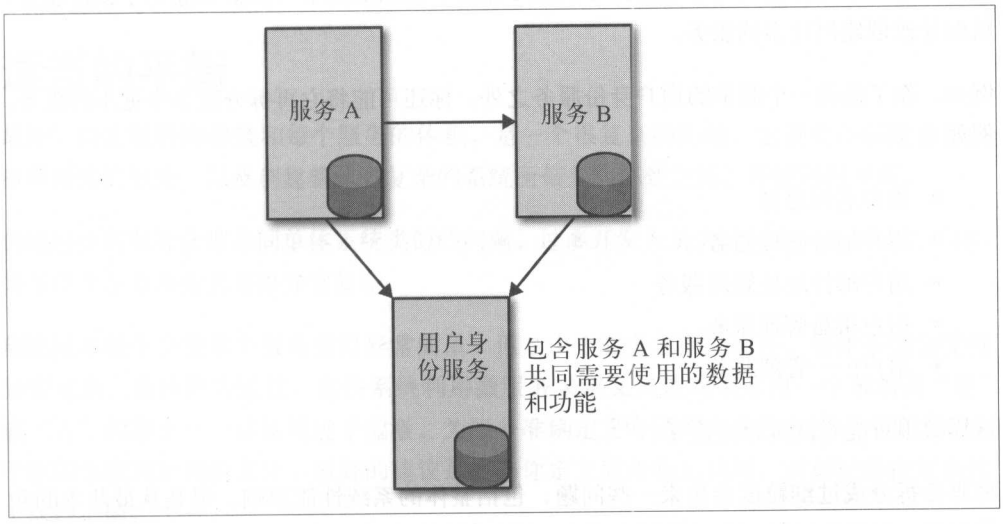


图12-3：通过服务与其他服务共享通用数据

这个数据服务的业务逻辑也许不复杂，但是它最终负责所有用户的常用信息。这些信息通常被大量其他的服务所使用。 81

通过一个中心服务来提供和管理这类信息，在实际中非常实用。

多种原因

之前的指导原则大概描述了决定服务边界的基本标准。但是，通常你需要综合考虑多种原因才能最终做出决定。

例如，从数据所有权和共享能力的角度来看，只有一个用户身份服务是合理的，但是从团队所有权角度来看，可能就不一定了。保存用户身份的数据存储在某个数据库中可能是合理的，但是可能存储在一个或多个单独的服务中会更好。

举例说明，一个用户的个人资料中通常会有搜索选项，但是很少被搜索以外的功能使用。因此，这个数据可能更适合存储在一个搜索身份服务中，而不是用户身份服务。这也可

能是出于数据复杂性甚至是性能原因的考虑。[注1]

82 最终，你必须同时运用你的判断力和之前的标准。当然，你还必须考虑公司的业务逻辑和需要，以及特定的业务需求。

过犹不及

但是，通常你会将应用程序过度拆分成太多服务。之前的标准可能会被滥用于创建服务，从而导致创建出过多的服务。

例如，除了提供一个简单的用户身份服务之外，你还可能将它再拆分成多个更小的服务，例如：

- 用户名称服务
- 用户地址管理服务
- 用户邮件地址管理服务
- 用户祖籍管理服务
- 用户……管理服务

这样做很可能有点拆分过度了。[注2]

将服务拆分成过细粒度会带来一些问题，包括整体的系统性能影响。但是从最基本的角度来说，每当你要将一个功能拆分成多个服务时，你都会做以下几件事：

- 降低单个服务的复杂性（通常来说）。
- 增加系统整体的复杂性。

体积小的服务通常会降低单个服务的复杂性，但是，你拥有的服务越多，需要交互的独立服务也越多，从而导致整体系统架构变得越来越复杂。

如果一个系统中有过多的服务存在，可能会带来以下这些问题。

影响大局观

想要记住整个系统架构变得越来越难，因为系统变得越来越复杂。

83 更大故障几率

更多独立组件之间需要相互工作，导致故障发生的几率更大。

注1 如果搜索选项只在某些特定场合中使用，为什么每次获取用户身份信息的时候都要包含搜索选项呢？

注2 好吧，忘掉“很可能”吧，这已经“很显然”是过度拆分了。

难以修改服务

每个独立服务的消费者都越来越多，增加了修改服务可能对使用者造成影响的可能。

更多依赖

每个独立服务都越来越依赖于其他的服务。更多的依赖意味着发生更多问题的可能。

许多这些问题，都可以通过在服务间定义清晰的接口边界来缓解，但是这不是一个完美的解决方案。

适当的平衡

最终，决定服务的数量和每个服务的体积，是一个很复杂的问题。它需要在创建更多服务所带来的好处，以及创建整体更复杂的系统所带来的坏处之间，仔细进行平衡。

创建过少的服务会带来同单体系统类似的问题，过多开发人员会同时在一个服务上工作，并且单个服务本身会变得异常复杂。

创建过多服务会使单个服务变得异常简单，但是由于服务之间的交互，整体系统会变得异常复杂。我真的听说过，有些系统利用微服务来定义只是简单返回一个布尔值“是”或“否”的服务——这显然过于偏激。为服务准确定义合适的大小没有错，但是这取决于你的系统和公司的文化。最好的建议是，当你定义服务和架构时，时刻记得在复杂性上保持平衡。

在系统、组织和公司文化之间找到适当的平衡，对于充分利用微服务的好处非常重要。

85

第 13 章

处理服务故障

在构建大型基于微服务的系统时，如何处理服务故障是一个必须要解决的问题。你拥有的服务越多，服务出现故障的可能性越大，依赖于故障服务的其他服务数量也会越多。

级联式的服务故障

假设你有一个服务，它依赖于多个服务，并且有一些服务也依赖于它。图 13-1 展示了这个“我们的服务”与它的多个依赖（服务 A、服务 B 和服务 C），以及依赖于它的多个服务（消费者 1 和消费者 2）。

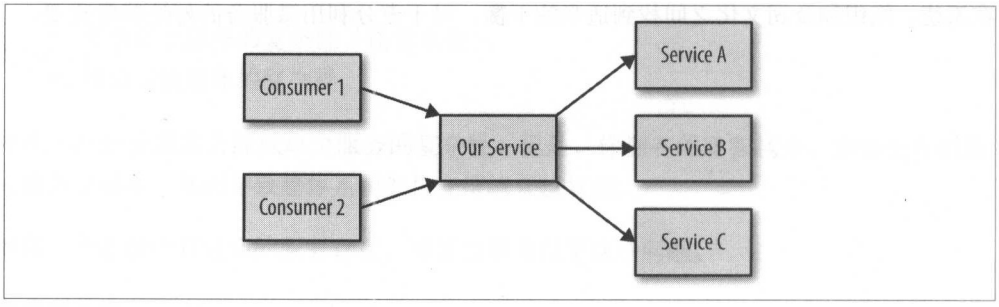


图13-1：我们的服务，以及它的依赖服务和消费者们

如果其中一个依赖服务出现了故障会怎样呢？图 13-2 展示了服务 A 出现故障时的情形。

除非你非常小心，否则这可以导致“我们的服务”也出现故障，从而又导致消费者 1 和消费者 2 出现故障。这个错误是可以级联发生的，如图 13-3 所示。

如果没有经过仔细检查，系统中的一个服务可能会导致整个系统发生严重的问题。

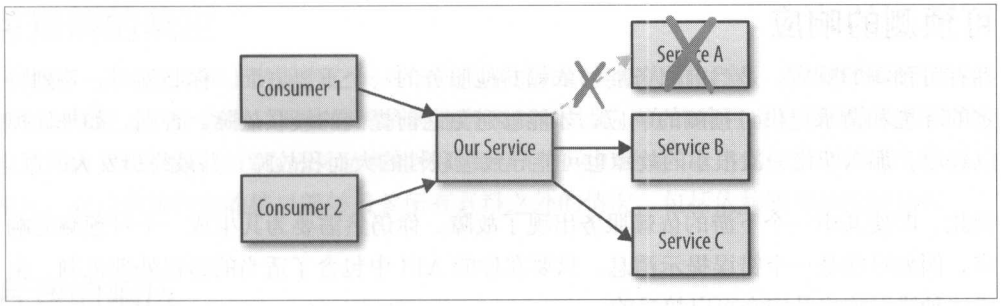


图13-2：我们的服务，和一个发生故障的依赖服务

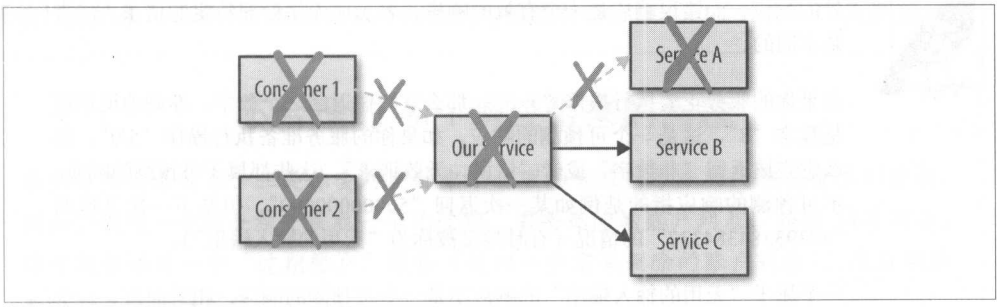


图13-3：级联故障

如何防止级联故障的发生呢？有些时候你什么也做不了——某个依赖服务中的一个错误就会导致你的服务（以及其他依赖服务）出现故障，因为这是从依赖的角度所决定的。有些时候，如果依赖服务发生故障，你的服务就无法完成应有的工作。但也不是全部如此。事实上，当某个依赖服务出现故障时，你还是有很多措施能够挽救自己的服务。在本章中，我们将讨论这些技巧。

86

如何响应服务故障

87

当你的某个依赖服务出现故障时，你应当如何响应？作为一个服务的开发人员，你对依赖服务故障的响应必须是：

- 可预测的
- 可理解的
- 对当前情形是合理的

我们来分别讨论一下其中的每一点。

可预测的响应

拥有可预测的响应，是当前服务能够依赖其他服务的一个重要方面。你必须对一系列特定的环境和请求提供可预测的响应，才能够避免之前提到的级联故障。否则，如果你掉以轻心，那么即使一次很小的故障也可能导致级联性的大面积故障，并最终引发大问题。

因此，即使其中一个下游的依赖服务出现了故障，你仍然需要为其生成一个可预测的响应，例如可能是一个错误提示消息。只要在你的 API 中包含了适当的错误处理机制，生成这种错误响应是完全可以接受的。

错误响应与不可预测的响应并不一样。一个不可预测的响应指的是服务预料之外的响应，而错误响应是一个有效的响应，表示你无法处理特定的请求。二者是不同的。

如果你的服务正要执行操作“3+5”，那么它应该返回一个数字，准确地说应该是数字“8”，这是一个可预测的响应。如果你的服务准备执行操作“5/0”，那么它应该返回“非数字”或者“错误，无效请求”。这些都属于可预测的响应。不可预测的响应指的是例如某一次返回“50000000000”，但是下一次又返回“38393384384337”的情况（有时候又被称为“无用的输入输出”）。

一个属于“无用的输入输出”的响应不是一个可预测的响应，相对而言，一个可预测的响应应该例如“无效的请求”。

88 你的上游依赖服务会希望你提供一个可预测的响应。当你遇到无用输入时，不要也返回一个无用输出。如果你将一个不可预测的响应提供给了下游服务，那么会在整个价值链上传递这种不可预测性。迟早这种不可预测的行为会反馈到你的用户那里，从而影响业务的发展。可能更糟的是，这种不可预测的响应会将无效的数据插入到你的业务流程中，导致业务流程数据不一致和数据无效。这会影响到你的业务分析，也会给用户造成不好的体验。

即使你的依赖服务出现故障或者发生了不可预测的行为，你也尽可能不要把这种不可预测性传递给依赖于你的服务。

可预测的响应实际上意味着它是一个计划中的响应。不要有“好吧，如果依赖服务出现故障，我也做不了任何事情，只好任由服务出现故障”这样的想法。相反，如果所有事情都出现故障了，你需要主动发现在当前情况下，一个合理的响应应当是什么样的，然后再检测当前情况是否满足条件，并返回预期的响应。

可理解的响应

可理解，意味着你和上游服务之间对响应的格式和结构都表示同意，从而在你和上游服务之间形成了一个约定。即使你的依赖服务出现了故障，你的响应也必须控制在约定的边界之内。永远不应该仅仅因为依赖服务违反了它的API约定，就违反你自己的API约定。相反，应该确保约定的接口能够覆盖所有意料之外的情况，包括依赖服务故障的情况。

合理的响应

你的响应应该说明当前服务实际发生了什么事情。当问到“3+5 等于几？”的时候，即使依赖服务出现故障，也不应该返回内容为“红色”的响应。它可以返回“对不起，我无法计算结果”或者“请稍后尝试”的响应，但绝对不应该返回“红色”作为答案。

不合理的 API 响应带来的问题

虽然听上去没什么，但是你会对实际中不合理的响应所带来的问题数量感到惊讶。例如，假设一个服务希望获取一个已过期、需要删除的账户列表，如图 13-4 所示。你可能会调用一个“过期账户”服务（返回一个需要删除的账户列表），然后继续删除列表中的所有账户。

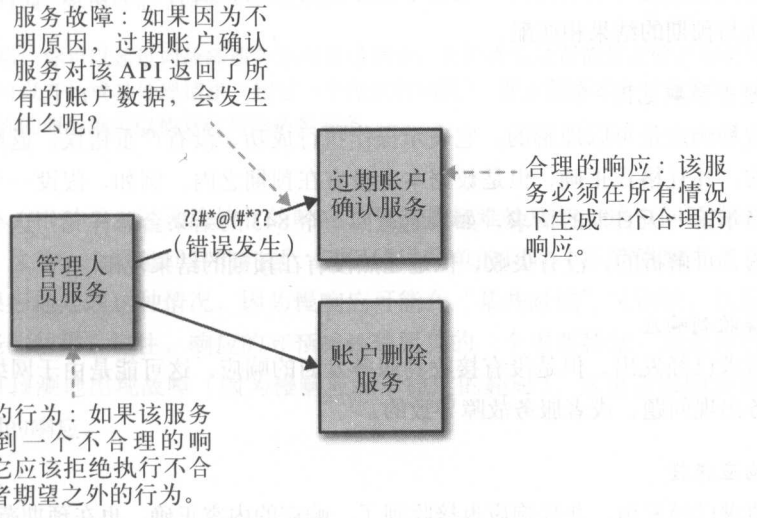


图13-4：不合理的API响应

如果“过期账户”服务出现问题，无法返回有效的响应，它应当返回“无”，或者“对不起，现在无法返回结果”这样的响应。想象一下，如果它没有返回一个合理的响应，而是返回了系统中所有账户的列表——那么你可以想象到后果了。[注1]

如何确定故障

如何确定一个依赖服务是否出现故障呢？这取决于故障的模式。以下是几种需要重点了解的故障模式。

乱码响应

这种响应是无法理解的。它是一种以无法理解的形式存在的“垃圾”数据。这可能代表响应的格式错误，或者格式中存在语法错误。

表示致命错误发生的响应

这种响应是可以理解的。它表示在处理请求时发生了严重的错误。这通常可能不是因为网络传输的原因，而是服务本身出现了错误。它还可能是由于发送给服务的请求无法被理解而引起的。

结果可以理解但是所需的结果不匹配

这种响应是可以理解的。它表示操作执行成功，没有严重错误，但是返回的数据无法与预期的结果相匹配。

结果超出预期范围

这种响应是可以理解的。它表示操作执行成功，没有严重错误。返回的数据是合理的，并且格式正确，但是数据本身没有在预期之内。例如，假设一个计算当天距离当年第一天日期的请求，如果它返回一个 843 的结果会怎样呢？这个结果是可理解的、可解析的，没有失败，但是显然没有在预期的结果范围之内。

没有接收到响应

请求已经发出，但是没有接收到服务发回的响应。这可能是由于网络通信故障、服务出现问题，或者服务故障导致的。

接收响应很慢

请求已经发出，并且响应也接收到了。响应的内容正确，也在预期范围之内。但是，响应比预期到达时间晚很长时间才接收到。这通常表明服务或者网络压力过大，或者存在其他资源分配不合理的问题。

注1 是的，这种问题真的在实际系统中出现过，而且，这真的很可怕。

以上这些按照检查程度从易到难排列。当你接收到乱码时，你立即知道响应不可用，并且可以采取适当行动。一个可理解但是不匹配所需结果的响应可能更难被发现，也更加难以决定如何行动，但是至少我们还可以得到这些响应。

一个永远无法到达的响应，使你很难对结果采取适当的行动。如果你只想给用户返回一个错误响应，那么在依赖服务处加上一个简单的超时处理，可能就能够捕获到丢失的响应。

捕获永远无法到达的响应的更好方式

91

但是，这不一定总是行得通。例如，如果一个服务通常需要 50 毫秒响应请求，但是有些时候可能只需要 10 毫秒，但有些时候又需要 500 毫秒，你该怎么办呢？你应该将超时设置成多少呢？一个显然的答案是大于 500 毫秒。但是，如果你对用户承诺的响应时间小于 150 毫秒呢？显然，仅仅将时间设置成 500 毫秒并不合理，因为这就相当于将依赖服务的错误直接传递给了用户。这违反了可预测和可理解的测试原则。

你如何来解决这个问题呢？一个可能的答案是使用断路器模式。这种代码模式让你的服务可以追踪对依赖服务的所有调用，了解到多少次调用成功，多少次调用失败（或者超时）。如果失败次数达到某个阈值，断路器会“切断”与依赖服务的连接，让你的服务以为依赖服务已经宕机，并停止向该服务发送请求或者接收响应。这使得你的服务可以立即检测到依赖服务发生故障并采取适当的行为，从而保证上游服务的 SLA。

随后，你可以通过向故障服务定期发送请求，来检查它是否恢复正常。如果它又可以开始成功处理请求（超过一个预定的阈值），那么断路器会“重置”，此时你的服务就可以恢复使用该依赖服务。

一个很慢的响应（相对于永远无法接收的响应）可能是最难处理的。这个问题变成了到底多慢算是慢？这是一个很难回答的问题，仅仅通过简单的超时（不管使用或者不使用断路器）很难很好地处理这种情况，因为慢响应可能在“某些时候”又很快，从而导致出现一些很古怪的结果。记住，响应的可预测性是服务的一个重要特征，如果你所依赖的服务总是不可预测地出现故障（因为慢响应和不稳定的超时），这也会影响你对依赖服务提供可预测的响应。

92

一种检测慢依赖的更优雅办法

除了断路器和相似的方法之外，还有一种更优雅的超时机制可以处理这种情况。例如，你可以创建多个“桶”来捕获最近调用依赖服务的性能。每次调用依赖服务时，你将返回响应的时间存储到一个桶中。你只在桶中保存一段时间内的响应时间。然后，可以通过这些桶来计算出一个触发断路器的规则。例如，你可以创建以下这些规则：

- 如果接收到“1 分钟之内 500 个请求超过 150 毫秒”，则触发断路器。
- 如果接收到“1 分钟之内 50 个请求超过 500 毫秒”，则触发断路器。
- 如果接收到“1 分钟之内 5 个请求超过 1000 毫秒”，则触发断路器。

这类分级技巧，可以帮助你更早发现更严重的慢响应问题，又不会忽略轻微的慢响应问题。

适当的行为

当错误发生时你应该做什么呢？这取决于错误的类型。以下是一些针对不同错误类型你可以采取的方案。

优雅降级

如果一个依赖服务出现了故障，你的服务没有响应还能够工作吗？它是否能够在缺少故障服务的响应下，继续执行所需的任务？如果这种情况下它还能执行有限的功能，那么就是一个优雅降级的例子。

优雅降级指的是在当前服务缺少某个故障服务的结果时，可以通过降低工作量来尽可能地完成工作。

示例13-1：提供有限功能

假设你有一个 Web 应用程序，用来创建一个售卖 T 恤的电子商务网站。我们还假设有一个“图片服务”为网站上显示的图片提供 URL。如果应用程序调用了该图片服务，但是图片服务出现了故障，那么应用程序应该怎么办呢？一种选择是，应用程序继续给用户
93 显示所需产品，但是没有该产品的图片（或者显示一条“图片不可用”的消息）。这个 Web 应用程序可以继续作为电子商务网站运行，只是缺少了显示产品图片的能力。

这种做法，比起只是因为图片不可用，整个电子商务网站就只好停运并返回错误的做法已经好太多了。

这就是一个提供有限功能的例子。

对于一个服务（或者应用程序）来说，即使因为依赖服务由于故障原因不能再提供所需数据，也应当尽可能地提供服务价值，这一点是非常重要的。

优雅补偿

有观点认为如果请求没有返回足够有用的结果，那么也应该算作失败。除了生成一条错误消息之外，你还能做些什么为用户提供价值呢？

示例13-2：优雅补偿

我们继续示例 13-1 中所描述的请求，假设提供指定产品详情内容的服务失败了。这意味着网站无法为所需产品提供任何可显示的信息。显然只显示一个空白页是不合理的，因为这样对用户来说没有任何用处。显示一条错误消息（“对不起，发生了一个错误”）也不是一个好主意。

相反，你可以显示一个向用户道歉但同时提供网站上当前最流行产品链接的页面。虽然这并不是用户真正想要的，但是它给用户提供了一些价值，避免了仅仅显示一个突兀的错误页面。

即使你无法完全满足用户的需要，但是按照为用户提供价值的方向去改进，这就是优雅补偿。

尽早失败

如果你的服务无法收到故障服务的响应，就无法工作呢？如果无法选择提供有限的功能或者优雅补偿呢？当你无法接收来自故障服务的响应时，已无计可施。在这种情况下，你能做的只有让请求失败了。

如果你已经确定无法挽救请求，那么应当尽快让请求失败。当你知道请求一定会失败后，就不要再执行请求中其他的操作或任务。

这样做的结果，就是你会尽可能对请求进行完备的检查，尽可能提前确认请求的正确性，当你继续发送请求时，应当有很大把握请求是可以成功的。 94

假设有一个计算两个整数相除的服务。我们都知道除数是不能为 0 的。如果你接到一个像“3/0”这样的请求，你可能会试着计算结果。在计算过程中，你最终会发现无法得到结果，于是抛出一个错误。

既然你知道所有除以 0 的计算都会失败，所以只需对请求的数据进行一下检查。如果除数是 0，则立刻返回一个错误。没有任何理由再去尝试进行计算。

为什么应该尽早失败？有以下几个原因：

资源节约

如果请求注定失败，那么在它失败之前你的所有工作都是白费。如果这些工作包括需要多次调用依赖的服务，那么浪费这么多宝贵的资源却只能得到一个错误。

响应性

你越快确定一个请求会失败，你就能越快地把这个结果返回给发起请求的一方。这使得请求方可以更快地做出其他决定。

错误复杂性

有些时候，如果你继续处理注定失败的请求，很可能会造成更难诊断的问题。我们还以“3/0”这个计算为例，你可以立即确定这次计算会失败，并返回错误消息。相反，如果你选择继续计算，就会产生错误，但是可能会以更加复杂的方式出现——例如，根据你计算除法的算法不同，这可能会导致一个无限循环，最终只有等到超时后才能退出。[注2]

因此，除了提示一个例如“除以 0”的错误之外，你还可以等待很长时间并得到一个“操作超时”的错误。相比而言，你应该很清楚哪个错误信息在诊断问题时会更加有用。

95 用户导致的问题

当你的服务可以接收来自用户的无效输入时，尽早失败变得更为重要。如果你知道服务已经规定了哪些合理的限制条件，请尽早检查它们。

一个说明“尽早失败”重要性的真实案例

在我之前工作过的一家公司中，有一个账户服务存在性能问题。该服务逐渐变得越来越慢，直到它几乎不可用。

在深入研究问题之后，我们发现有人向账户服务发送了一个错误请求。有人请求账户服务去获取 100,000 个用户的账户列表，包括所有账户的详细内容。

现如今，没有任何正常的业务会有这样的需要（在这个环境中），因此这个请求本身显然是无效的。100,000 这个值超出了这个请求的合理范围。

但是，账户服务很负责任地尝试去处理这个请求，于是处理、处理、不断地处理……

这个服务最终由于没有足够资源来处理如此巨大的请求，所以失败了。当处理了几千个

注 2　除法操作的一个经典算法，是使用连续的减法操作来实现除法。除非提前发现错误，否则除以 0 会导致该算法形成一个无限循环。

账户后它停止了服务，并返回了一个简单的错误消息。

而发起无效请求的调用方服务，由于接收到了失败消息，于是决定重试这个请求，于是重试、重试、不断地重试。

账户服务不断地处理这几千个账户，将这些结果扔到了一个失败信息中。但是它不断在重复做这件事。

不断失败的请求消耗了大量的可用资源。随着过多资源被消耗，导致其他合法请求也开始排队，并最终导致整个服务出现故障。

如果能在账户服务处理请求前进行一个简单的检查（例如检查请求的账户数量是否合理），就可以避免对资源的大量浪费。此外，如果返回的错误消息能够表明这个错误是永久性的，并且是由一个无效的参数所导致的，那么调用方服务能够看到这个“永久错误”的消息，也就不会再去重试一定会失败的请求。

96

提供服务约束

这个案例的结果告诉我们一定要提供服务约束。例如，如果你知道服务不能同时处理超过 5000 个账户，一定要在服务约定中声明这个限制条件，进行测试，让所有超过该限制数量的请求都直接失败。

如何让应用程序具有伸缩性

即使是海螺也能够伸展。

99

第 14 章

两次失误的高度

我想先和你分享一个我无意中听到的故事：

> 我们正在考虑修改 MySQL 数据库上的一个设置会对性能有什么影响，但是担心这个改动会导致生产数据库发生故障。因为我们不想弄垮生产数据库，所以决定将这个改动先应用到备份（复制）数据库上。毕竟，备份数据库并不是有人一直在使用。

听上去有点道理，是吧？不知道你之前有没有听到过这样的事情？

好吧，这里的问题在于数据库正在被人使用着。它目前的用途是为生产数据库提供备份，除此之外，它不应当被用作其他任何用途。

如你所见，出于实验性质的目的，备份数据库现在被用于测试不同的设置。这样带来的结果就是随着这些设置生效，备份数据库开始与主生产数据库的差异越来越大。

直到，有一天，不可避免的事情发生了。

生产数据库出现了故障。

备份数据库一开始的确做了应该做的事。它接过了主数据库的工作。只是事实上它做不到。由于备份数据库上的设置已经与主数据库上的设置相差太多，导致它无法再正常处理原来主数据库上的数据。

于是备份数据库慢慢发生了故障，然后整个网站都无法工作了。

100 这是一个真实的故事。你有一个备份、复制的数据库，原本在主数据库发生故障时，准备作为主数据库来使用的。但是，这个备份数据库没有受到像主数据库一样的尊重对待，并且也失去了它的主要能力——成为备份数据库的能力。

两次失误不会让事情变得正确，两次失误也无法抵消彼此，两次失误也无法自己修复。一次主数据库故障再加上一个管理不善的备份服务器，会让这一天变得糟糕。

什么是“两次失误的高度”

如果你曾经操作过无线电控制（R/C）飞机，可能听说过那句话“让你的飞机保持两次失误的高度”。

当你学习如何操作 R/C 飞机，尤其是开始学习如何进行特技飞行时，你会学得很快。失误其实就相当于高度，出现一个失误，你就失去一定的高度。当你失去太多高度时，坏事就发生了。因此，让你的飞机保持“两次失误的高度”意味着让你的飞机飞得足够高，从而有足够的高度从两个不相关的失误中恢复飞行。

为什么是两次失误？这很简单。你总是希望让飞机飞得足够高，以便可以从一次失误中恢复飞行。现在，假设你犯了一个失误，失去了一些高度。在从这个失误的恢复过程中，你还会希望飞得足够高，从而保证能够从另一次失误中恢复飞行。你可以想象一下：在恢复飞行的过程中，你通常压力很大，而且可能处于一种惊恐的情绪中，很可能会做出一些反常的事情——正是这种情形让你可能产生另一个失误。因此如果你飞得不够高，就会坠机。

换一种角度说，如果你能够飞行两次失误的高度，你总有一次从失误中恢复的备份方案，即使你正在从一次失误中恢复。

同样的思想，对于我们创建高可用、大规模的应用程序是非常重要的。

我们如何在应用程序中“保持两次失误的高度”？对于初学者而言，当我们确定出系统要面对的失败场景后，我们遍历所有可能存在的分支场景，并制订相应的恢复计划。我们需要确定恢复计划本身没有错误或者缺陷——简而言之，需要检查恢复计划是否可以正常工作。如果发现它不能正常工作，那么就应当重新制订恢复计划。

实践中的“两次失误的高度”

101

这只是一个可能会用到“两次失误的高度”的场景，当然还有很多其他的场景。我们会通过一些示例来说明它对应用程序的影响。

丢失一个节点

我们来看一个跟 Web 服务流量有关的示例场景。

示例14-1：需要多少个节点

假设你正在开发一个预计每秒处理 1000 个请求（请求 / 秒）的服务，而且我们假设服务中的单个节点每秒只能处理 300 个请求。

问题：你需要多少个节点才能支撑该流量？

通过一些简单的数学计算我们可以得到结果：

$$所需的节点数 = 请求总数 / 每个节点能处理的请求数量$$

其中：

所需的节点数
: 表示处理指定数量请求所需的节点数量。

请求总数
: 该服务预期处理的请求总数。

每个节点能处理的请求数量
: 服务中每个节点能处理的请求数量的平均值。

代入我们的数字之后：

$$所需节点数量 = 1000 / 300 = 3.3 = 4 个节点$$
$$所需节点数量 = 4 个节点$$

你需要 4 个节点来处理每秒 1000 个请求的服务压力。转换一下，当使用 4 个节点时，每个节点需要处理：

102

$$每个节点请求数量 = 请求总数 / 节点数量$$
$$每个节点请求数量 = 1000 / 4 = 250 请求 / 秒 / 节点$$

每个节点需要每秒处理 250 个请求，这低于每秒 300 个请求的单个节点限制。

现在你有了 4 个节点。你不仅可以处理预计的流量，并且因为有了 4 个节点，可以允许丢失掉一个节点。你已经做好丢失一个节点的准备了，是不是？真的吗？

当然，实际上你并没有。如果你在流量峰值时丢失一个节点，你的服务仍然会失败。为什么？因为如果丢失一个节点，你的流量会分布在剩余 3 个节点上，如示例 14-2 中所示，这样是行不通的。

示例14-2：丢失一个节点

在示例 14-1 的系统中，如果丢失了 4 个节点中的 1 个，只剩下 3 个节点来处理流量。此时：

$$\text{每个节点请求数量} = \text{请求总数} / \text{节点数量}$$

$$\text{每个节点请求数量} = 1000 / 3 = 333 \text{ 请求 / 秒 / 节点}$$

每个节点需要每秒处理 333 个请求，这超过了每秒 300 个请求的节点限制。

因为每个节点每秒只能处理 300 个请求，所以服务器现在已经过载。在当前情况下，或者给所有用户提供较差的性能，或者会丢掉某些请求，或者以其他形式无法提供服务。总之，你的可用性开始下降了。

如你所见，如果系统失去了其中的一个节点，就无法继续提供完整的能力。因此，即使你以为能够从一个节点故障中恢复，但实际上却不能。你的系统现在是脆弱的。

为了能够处理节点故障，你需要 4 个以上的节点。如果你希望能够处理一个节点故障，那么就需要 5 个节点。这样，即使 5 个节点中的其中一个出现故障，还剩下 4 个节点可以处理流量，如示例 14-3 所示。

示例14-3：丢失一个备用节点

103

如果你丢失了 5 个节点中的其中 1 个，还剩下 4 个节点，那么每个节点需要承担的流量是：

$$\text{每个节点请求数量} = \text{请求总数} / \text{节点数量}$$

$$\text{每个节点请求数量} = 1000 / 4 = 250 \text{ 请求 / 秒 / 节点}$$

因为这个值低于每个节点每秒 300 个请求的限制，所以它们有足够的能力来继续处理流量，即使是一个节点出现了故障，也不会对系统造成任何影响。

升级过程中出现的问题

升级和日常维护可能会导致未预计的可用性问题。我们以示例 14-4 来说明。

示例14-4：升级你的应用程序

假设你有一个平均流量为 1000 请求 / 秒的服务。此外，我们假定服务中单个节点的处理上限是 300 请求 / 秒。如示例 14-1 中所述，最少需要 4 个节点来运行服务。为了能够处理预期流量并支持单节点故障，你为服务配置了 5 个节点。

现在，假设你希望对正在运行的服务进行一次软件升级。为了在升级过程中保证服务正常运行，你决定使用轮流部署的方式。

简而言之，轮流部署就是每次只升级一个节点（临时将它设置为离线状态来进行升级）。在成功升级第一个节点并重新处理流量之后，继续升级第二个节点（临时将它设置为离线）。持续这个过程直到 5 个节点都完成升级。

因为在每次升级时，只有一个节点是离线状态，所以总是有 4 个节点在处理流量。因为 4 个节点已经足够处理所有的流量，所以升级过程中服务不会受到影响。

这是一个很不错的计划。你已经构建了一个不仅能处理单节点失败，还可以通过轮流部署实现不停机升级的系统。

如果在升级过程中某个节点发生了故障呢？这个时候，你有一个节点因为升级不可用，还有一个节点出现故障。这样只剩下 3 个节点，不足以处理所有的流量。这时，你就会遇到服务降级或者系统整体不可用。

104 在升级时遇到某个节点故障的可能性有多大呢？

你有多少次遇到过升级失败呢？事实上，升级过程中一个参数带来的节点故障几率，可能比其他时候大得多。升级和节点故障并不是完全孤立的。

我们得到的教训是：即使你认为有冗余节点来处理各种故障情况，但是如果两个或多个问题同时发生（因为问题都是有关联性的），就可能会根本没有任何冗余。这样的系统就容易出现可用性的问题。

总而言之，对于示例 14-1 中的系统，要想使用 300 请求 / 秒的节点来处理 1000 请求 / 秒的流量，我们需要：

4 个节点
: 可以处理流量，但是不能处理单个节点故障。

5 个节点
: 可以处理单个节点故障，或者在维护或升级时允许单个节点不可用。

6 个节点
: 可以处理多个节点故障，或者在维护或升级时，允许同时存在一个节点升级失败和一个节点不可用。

数据中心恢复

我们将这个问题扩大一点，来看一看数据中心的冗余和恢复。

示例14-5：一个更大的服务

假设你的服务正在处理 10,000 请求 / 秒的流量。因为单个节点只能处理 300 请求 / 秒，所以这意味着你需要 34 个节点，这还不考虑故障冗余和升级的情况。

为了让系统增加一些额外的处理能力，我们使用了总共 40 个节点（每个节点处理 250 请求 / 秒）。现在即使失去多达 6 个节点也可以处理所有的流量。

让我们来做得更好一点：现在我们把这 40 个节点平均分布到 4 个数据中心，这样可以有更好的冗余性。

现在，我们可以像恢复节点故障一样来恢复数据中心故障了，是这样吗？

这是一个好问题。显然，我们可以处理单个节点故障，因为我们已经提供了额外的 6 个（40-34）节点。但是如果某个数据中心出现故障了怎么办？ <105>

如果某个数据中心失败，我们就丢失了四分之一的服务器。在这个例子中，我们就从 40 个节点下降到了 30 个节点。此时每个节点不再只处理 250 请求 / 秒，而是需要处理 334 请求 / 秒。因为这超过了单个节点的处理能力，所以我们又遇到了可用性的问题。

即使我们使用多个数据中心，但是一旦某个数据中心出现故障，就会让我们无法再处理增长的流量。我们认为自己可以从某个数据中心的故障中恢复，但是实际上我们不能。

因此，你究竟需要多少台服务器？

究竟需要多少台服务器才能处理数据中心出现故障的情况？我们来一起解答这个问题。

依然基于示例 14-5 的相同假设，即我们需要最少 34 台服务器才能处理所有流量。如果我们使用 4 个数据中心，究竟需要多少台服务器才能真正满足数据中心冗余呢？

显然，即使 4 个数据中心其中之一出现故障，我们需要确保任何时候都拥有 34 台正常工作的服务器。这意味着我们需要有 34 台服务器遍布在其他三个数据中心中：

每个数据中心的节点数 = 服务器最小数量 /（数据中心数量 -1）

每个数据中心的节点数 = 34 /（4-1）

每个数据中心的节点数 = 11.333 = 12 台服务器 / 数据中心

因为我们需要在每个数据中心有 12 台服务器，并且即使 4 个数据中心之一出现故障，我们在每个数据中心仍然有 12 台服务器，所以：

总节点数 = 每个数据中心的节点数 * 4 = 48 个节点

因此，我们需要 48 个节点来保证，即使有一个数据中心出现了故障，依然有 34 个节点能够工作。

如果数据中心的数量发生了变化，会对我们的计算造成什么影响呢？我们来看一看示例 14-6。

示例14-6：不同数量的数据中心

如果我们只有两个数据中心呢？如之前一样：

106

每个数据中心的节点数 = 服务器最小数量 /（数据中心数量 -1）

每个数据中心的节点数 = 34 /（2-1）

每个数据中心的节点数 = 34

总节点数 = 每个数据中心的节点数 * 2 = 68 个节点

如果我们有两个数据中心，则需要 68 个节点。那其他时候呢，如果你拥有：

4 个数据中心

我们需要 48 个节点来保证数据中心冗余。

6 个数据中心

我们需要 42 个节点来保证数据中心冗余。

这说明了一个看上去很奇怪的结论：为了提供整个数据中心故障恢复的能力，当你拥有的数据中心越多，每个数据中心需要的节点越少。这与我们的直觉相差甚远。

从这个例子中，我们得到的教训是：虽然其中的内容可能无法直接应用到实际情况中，但是理论依然适用。当你在设计恢复计划时一定要小心，你的直觉可能会与实际情况相背离，如果你的直觉是错误的，那么就会带来可用性的问题。

隐蔽的共享故障类型

有些时候，很多问题看上去都是独立的，不太可能同时发生，但是事实上却是互相关联的。这意味着在某些场景下，多个故障可能会一起出现。

示例14-7：机架故障

假设你的服务运行在 4 个节点上。你为了做好充分准备，使用了一共 6 个节点——足够同时处理单节点故障和升级失败。

你现在放心地认为系统是安全的。

随后发生了一件事：在数据中心的机房中，某个机架上的供电系统出现了问题，导致整个机架无法工作。

通常在这个时候，你会意识到所有 6 个服务器都放置在同一个机架上。你是如何发现的呢？因为所有 6 台服务器都下线了，同时你的服务也完全无法提供了。 107

所以也别提什么冗余了……

当你认为自己的系统是安全的时候，可能实际上并不是这样。我们知道不是所有问题都是完全独立的。在你的所有服务器之间，普遍存在着可能看不到或者至少是没发现的情况，那就是它们都共享着相同的机架和供电系统。

这里的教训是：仔细检查隐藏的共享故障因素，它们可能会导致你精心准备的计划付之东流，为系统带来可用性风险。

故障循环

故障循环指的是，当某个特定问题导致系统故障时，为了修复它，你需要或者不得不制造另一个更严重的问题。

解释故障循环最好的方式，是用下面这个跟服务器无关的例子。

示例 14-8：车库门

假设你生活在一个非常棒的公寓中，有一个封闭的车库来存放东西。哇！这真不错。

但是这里经常停电，于是你决定买一台发电机，以备不时之需。你买了发电机、汽油，然后把它们放在车库中。生活看上去很美好。

然后，当停电时，你准备去车库找你的发电机。

这时候你第一次意识到，唯一进入车库的方式就是通过车库的电子门——但是因为停电它无法打开。

哎呀！

这正是因为虽然你制订了备选计划，但并不意味着在需要时能执行它。

同样的问题也会出现在我们的服务上。一次服务故障会不会因为导致其他看起来无关的 108

问题，从而让我们难以修复？

例如，你的服务出现故障，部署一个新的版本有多简单？如果你用于部署的服务失败会怎样？如果你用来监控其他服务性能的服务失败了会怎样？

这里的教训是：确保你的恢复计划在问题发生时能够实现。如果不能考虑问题间的依赖关系以及相应的解决方案，你就会面临可用性的问题。

管理你的应用程序

“两次失误的高度”意味着不仅要看到表面的问题，还要往深层次了解。你需要确保不存在互相依赖的问题，并且准备的恢复机制能够在问题发生时真的帮你恢复系统。

此外，请不要忽视问题。问题不会自己消失，并且它们会根据你的可用性计划发生变化。因为即使出现故障的是备份数据库，也并不意味着放着对它不管。对待你的备份和冗余系统，应该像对待主系统一样认真，毕竟它们的重要性是一样的。

我经常跟朋友们说，“如果它跟生产环境有接触，那它就是生产环境”。不要认为生产环境中的任何事都是稳定可靠的。

做到这一点很难。我们很难知道什么时候会出现不同级别或相互依赖的故障。你应当多花一些时间来观察系统的情况并解决它们。

航天飞机

让我们用一个独立、冗余、能够恢复多级错误的系统示例来结束本章。事实上，它也是第一个将冗余和故障管理思想发挥到极致的大规模软件系统。它让宇航员可以将生命托付给它。

没错，我指的就是美国航天飞机计划。

航天飞机计划有一些重大且严重的机械问题，但是我们不会在这里来讨论它们。我们要说的是航天飞机中的软件系统，以及它如何将冗余和独立的错误恢复发挥到极致。

109 航天飞机的主计算机系统由 5 台计算机组成。其中 4 台运行的软件和硬件都一样，但是第 5 台则不同。我们稍后会来讨论这一点。

在执行任务的关键部分（例如发射和着陆）时，这 4 台主计算机都会运行一样的程序。

这 4 台计算机中的数据和软件也是一模一样的，因此它们计算出的结果也应该是一样的。所有这 4 台计算机执行一样的计算，并不断地与其他计算机比较结果。如果，在某一时刻，任意一台计算机计算出了一个不同的结果，这 4 台计算机会投票选举哪一个结果是正确的。在使用获得选举的结果后，产生错误的计算机会在整个飞行过程中被关闭。航天飞机可以在只有 3 台计算机的情况下安全飞行，并且可以在只有两台计算机的情况下安全着陆。

这就相当于民主制度的最终形态。胜者统治，败者终结。

但是如果 4 台计算机无法达成一致呢？这可能出现在多次失败以及多台计算机关机的情况下。或者，由于某个严重的软件问题同一时间影响了 4 台计算机（毕竟这 4 台计算机都运行着完全一样的软件）。

这时就需要第 5 台计算机来发挥作用了。通常它都处于空闲状态，但是如果有需要，它可以执行与其他 4 台计算机一样的计算。关键在于它上面运行的软件。第 5 台计算机上运行的软件，是由一个完全独立的开发小组开发的简化版本。在理论上，它不会与主软件存在一样的软件错误。

因此，如果 4 台计算机上的主软件之间不能就结果达成一致，它会将决定最终结果的权利交给第 5 台完全独立的计算机。

这就是一个从上层角度隔离可能问题的高冗余、高可用的系统。

在运行的 30 年中，航天飞机计划从来没有在执行过程中，由于软件或者计算机的故障而遇到危及生命的严重问题——即使它是当时空间计划使用得最复杂的软件系统。

111

第 15 章

服务所有权

我们在第 12 章中说过，服务必须由一个独立的开发团队负责并维护，但是我们当时并没有详细讲解其中的意义。在本章中，我们会介绍服务所有权的意义，以及如何来实践由独立团队负责的服务架构。

由独立团队负责的服务架构

什么是“由独立团队负责的服务架构（STOSA）”？ STOSA 对于由多个开发团队来负责并管理一个或多个系统服务的大型组织来说，是一个重要的指导原则。

拥有一个 STOSA 的系统和组织意味着什么？要达到 STOSA，你必须满足以下几个条件：

- 有一个基于服务或者微服务架构建立的应用程序。
- 有多个开发团队负责构建和维护这个应用程序。
- 程序中的所有服务必须分配给某个开发团队。
- 不允许有服务被分配给多个开发团队。
- 一个开发团队可能会负责多个服务。
- 开发团队负责管理服务的所有方面，从服务架构到设计再到开发、测试部署、监控和故障处理。
- 服务之间有清晰的边界，包括文档编写良好的 API 接口。

112

- 各服务负责维护之间的服务等级协议，开发团队负责对服务进行监控和报警。

一个基于 STOSA 的应用程序，就是一个所有服务都满足以上条件的应用程序。一个基于 STOSA 的组织，就是所有服务团队都满足以上条件，并且所有应用都是 STOSA 应用的组织。

通常来说，在一个基于 STOSA 的组织中，每个团队的大小是确定的（通常在 3 到 8 个

人之间）。如果一个团队太小，它无法有效管理某个服务。如果团队太大，管理团队的工作会太繁重。

图 15-1 展示了一个基于 STOSA 的组织，以及它如何来管理 STOSA 应用程序。

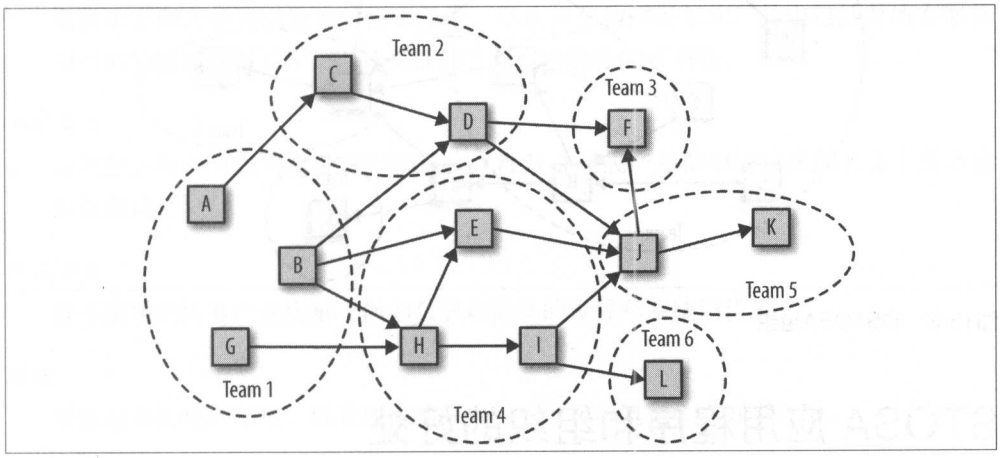


图15-1：管理STOSA应用的STOSA组织

在这张图中，标识从 A 到 L 的方块表示每个独立的服务。椭圆表示完全负责这个服务的开发团队。

这个应用系统包含了由 5 个团队管理的 12 个服务。你会注意到，每个服务由一个团队来管理，但是一个团队可能会管理多个服务。每个服务都有一个所有人，并且一个服务只能有一个所有人。

应用程序的每个方面都有清晰的所有权，对于系统的任何一个部分，你都可以很容易地知道谁是负责人，以及有疑问、问题或者改动应该联系谁。

图 15-2 展示了一个非 STOSA 的应用程序和组织。

你会注意到几件事情。首先，服务 I 没有所有者，并且服务 C 和服务 D 归属于多个团队。 113

在图 15-2 中，服务之间没有清晰的所有权。如果你需要在服务 C 或者服务 D 中完成某些事情，不清楚谁应该为此负责。如果其中一个服务出现问题，应该由谁来响应？如果你需要让服务 I 来完成某些事情，你应当联系谁？缺少清晰的所有权和责任会让一个复杂系统变得更加难以管理。

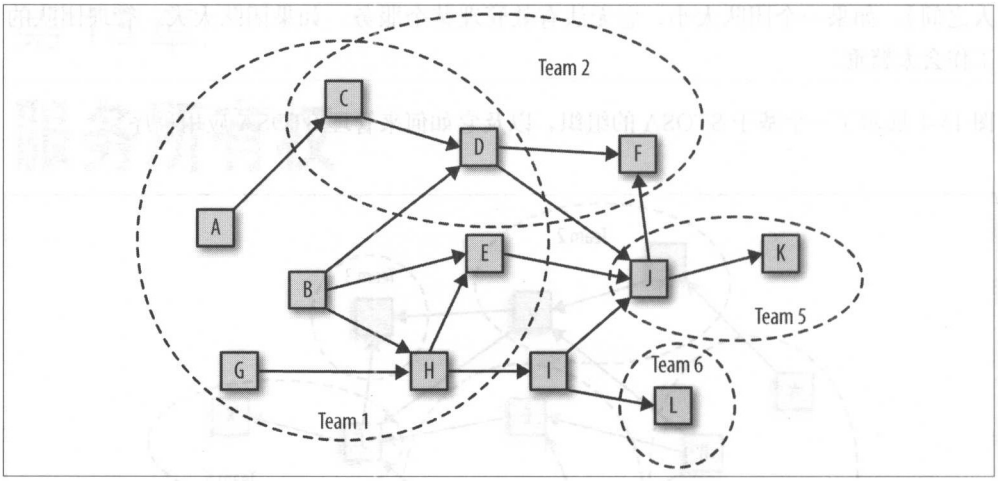


图15-2：非STOSA组织

STOSA 应用程序和组织的好处

随着程序体积的增加，复杂性也在增加。一个基于 STOSA 的应用程序，可以比一个非基于 STOSA 的应用程序发展得更大，可以由一个更大的开发团队来管理。因此，它可以在向更大规模发展的同时，依然保持稳定的、文档化以及可支持的接口。

一个基于 STOSA 的组织比一个非基于 STOSA 的组织，可以管理更大、更复杂的应用程序。这是因为 STOSA 可以在多个开发团队之间有效分散系统的复杂性，同时保持清晰的所有权和职责范围。

成为一个服务所有者意味着什么

在一个 STOSA 组织中，服务所属的团队需要 100% 负责服务的所有方面。这个团队可能需要其他团队的支持（例如硬件运维团队），但是最终该服务是由这个团队来完全负责的。

114 其中特别包括以下这些方面。

API 设计
: 负责所有内部、外部 API 的设计、实现、测试和版本管理工作。

服务开发
: 负责服务所有业务逻辑的设计、开发和测试工作。

数据

负责管理服务所产生和维护的所有数据，包括数据的展现、模型、访问方式以及生命周期。

部署

负责决定服务是否以及何时需要升级，以及相关的部署工作，其中包括对所有服务节点的验证和回滚工作，以及保证部署过程中服务的可用性。

部署窗口

决定什么时间可以安全地进行部署。这涉及公司或产品范围的停服以及各个服务的时间窗口。

产品变更

服务所需的所有产品变更（例如负载均衡器的设置和系统调优）。

环境

管理服务的生产环境，以及所有的开发、预发布以及预生产部署环境。

服务 SLA

讨论、确定并监控 SLA，以及保障服务满足 SLA 的相关工作职责。

监控

确保为服务的各个方面建立了监控，包括监控服务的 SLA。它还包括定期检查监控的情况。

值班 / 突发事件响应

确保系统在出现故障时能够生成报警事件。安排轮流值班制度以及报警管理，确保团队中有人能够处理突发事件。确保对突发事件的响应速度能够满足之前定义的 SLA。

报告

向其他团队（消费服务或者依赖服务）发送内部报告，以及管理服务的运行健康报告。

115

通常，以下几项不是由所属团队负责，而是由公共的基础设置、工具或者运维团队来负责。

服务器 / 硬件

所有生产和其他环境下需要的硬件和基础设施。这通常由一个运维团队、或者云服务、或者二者同时提供。

工具集

集中提供和管理团队需要的各种工具。这其中包括部署工具、监控工具、值班和突发事件响应工具，以及报告工具。

数据库

通常由一个公共团队来管理负责存储服务数据的硬件和数据库。但是，数据本身、数据模式以及对数据的使用，通常由服务所属团队来管理。

图 15-3 展示了一个基于 STOSA 的组织机构。特别要注意的是，从组织机构角度来看，所有服务所属的开发团队都是平级的。他们统一由一些支持团队，包括运维、工具、数据库以及其他团队来提供支持。所有这些团队可能都基于其他核心团队之上，为整个组织提供支持，而不是为某个单独服务提供支持。这可能包括像架构设计或者项目管理这样的团队。

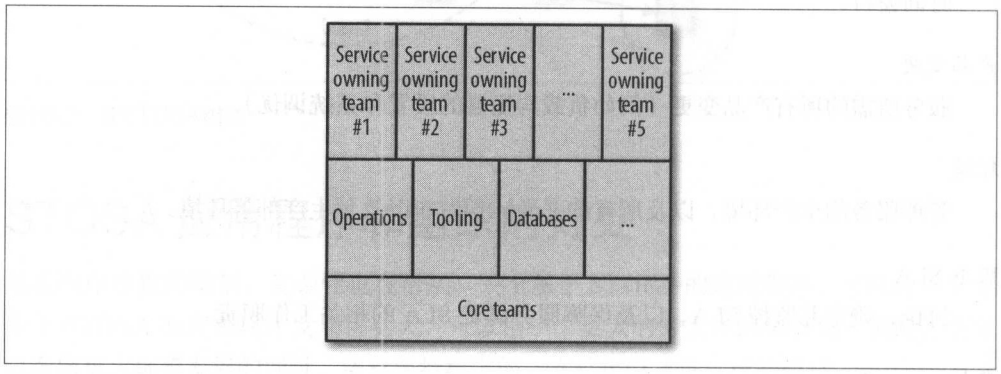


图15-3: 基于STOSA的组织架构

116 在 STOSA 组织中，服务所属团队会负责该服务的所有方面。他们可能会依赖于核心团队或者支持团队，但是最终是由他们来处理该服务的所有问题，保证服务的正常运行。

例如，我们假设因为核心团队的部署工具问题导致服务部署失败，那么也应该由服务所属的团队来负责。虽然他们可能需要与开发工具的团队来沟通解决，但是最终对此负责的一定是这个团队。他们不能托辞说“这是工具团队的错误”这样的话。因此，即使最后发现事实是这样，但是因为服务出现了故障，所以由负责这个服务的团队来承担责任。

伴随着对结果完全负责，服务团队拥有强大的决策权力。通常，服务所属的团队会被分配一系列需要实现的需求，但是如何实现需求的细节由他们来决定。这个团队可能需要遵守很多系统方面的要求（例如，架构方面的设计原则或者规则，必须使用的工具，或者只限使用某种编程语言和硬件），但是这些最终都只是团队需要满足的需求而已。

除了这些需求之外，所有的设计和实现细节都由服务团队来决定。

最终，服务所属团队会承诺达到预期的结果，并维护合理的 SLA。

第 16 章

服务分级

开发含有许多服务的大型、复杂的应用程序会存在可用性的问题。某个服务的故障可能会导致依赖它的所有服务都出现故障。这样所带来的级联影响会让你的整个系统都不可用，尤其是当一个非关键服务的故障导致了关键服务出现故障时，会带来非常恶劣的影响。

应用复杂性

如图 16-1 所示，有些时候，最不起眼的服务反而可能会出现故障。

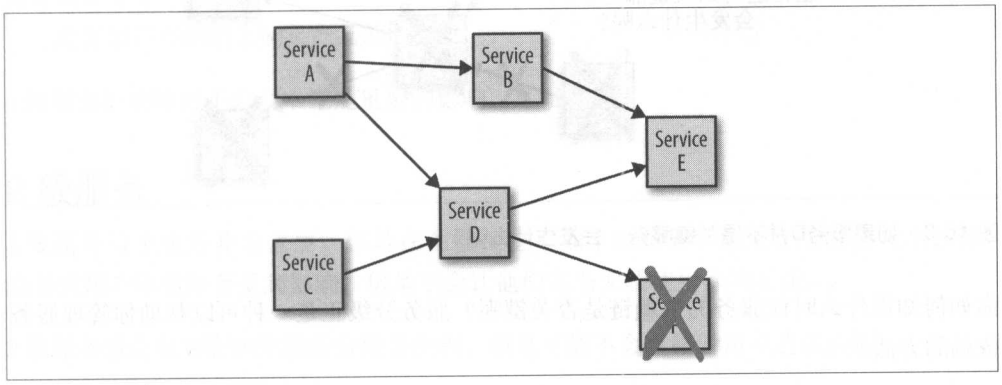


图16-1：某个服务失败……

这可能导致你的整个应用程序都无法提供服务，如图 16-2 所示。

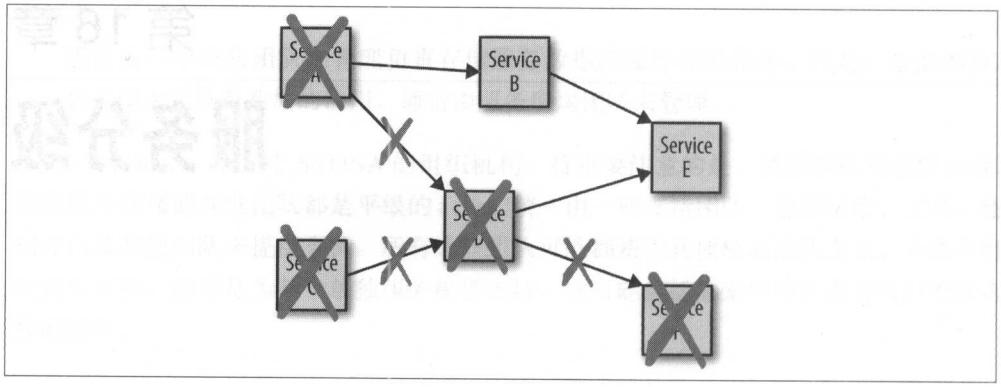


图16-2：可以导致级联性故障

118 我们已经在第 13 章讨论了很多可以预防依赖服务发生故障的方式。但是，服务间的弹性也会增加复杂性和成本，并且有些时候并不需要。以图 16-3 为例，如果服务 D 对于服务 A 的运行来说不是必要的，会发生什么呢？为什么仅仅因为服务 D 出现故障，服务 A 也要发生故障呢？

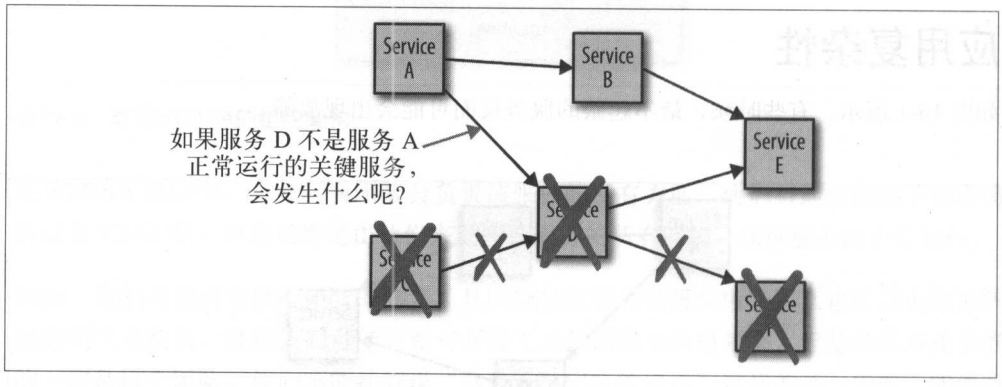


图 16-3：如果服务D并不是关键服务，会发生什么呢？

你如何知道什么时候服务的依赖链是否关键呢？服务分级正是一种可以帮助你管理服务依赖的方法。

什么是服务分级

服务级别其实只是与某个服务关联的标签，表示该服务对于业务运行的关键程度。服务
119 分级让你能够区分出哪些服务是关键型任务，哪些服务有用但不是至关重要的。

通过比较各个依赖服务的服务级别，你可以确定哪些服务依赖对于业务最为敏感，哪些

服务略微次之。

为服务分配服务级别标签

系统中的所有服务，不管它多大或多小，都应当被分配一个服务级别。以下章节会大致告诉你如何来为一个服务分配级别（你可以根据自己的业务需要来调整建议）。

1 级服务

1 级服务是系统中最关键的服务。如果某个服务出现故障会导致用户或者公司业务产生重大损失，那么应当考虑将这个服务定为 1 级。

以下是一些 1 级服务的示例。

登录服务
: 允许用户登录系统的服务。

信用卡处理服务
: 处理用户支付的服务。

权限服务
: 告诉你指定用户可以访问哪些功能的服务。

订单处理服务
: 允许用户在网站上购买产品的服务。

1 级服务的故障对于公司来说是重要的影响。

2 级服务

2 级服务对于业务非常重要，但是在关键性上不如 1 级服务。如果 2 级服务发生故障，会导致用户体验显著受到影响，但是不会让他们完全无法使用你的系统。

2 级服务也会显著影响你的后台服务流程，但是可能不会直接对用户造成影响。

以下是一些 2 级服务的示例。

搜索服务
: 为网站提供搜索功能的服务。

120

订单结算服务

让仓库能够处理订单以便将商品寄送给用户的服务。

2 级服务的失败会对用户产生负面影响，但是不会导致系统完全失败。

3 级服务

3 级服务会对用户造成较小的、不容易注意到的，或者很难发现的影响，或者对你的业务和系统造成有限的影响。

以下是一些 3 级服务的示例。

用户头像服务

在网站上显示用户头像图标的服务。

推荐服务

根据用户当前浏览的商品，显示他可能感兴趣的其他商品的服务。

每日提醒服务

在网页顶部显示用户提醒或者消息的服务。

用户甚至可能注意不到 3 级服务出现了失败的情况。

4 级服务

4 级服务即使失败，也不会对用户体验造成任何严重的影响，更不会对业务或者资金方面造成任何严重损失。

以下是一些 4 级服务的示例。

销售报告生成服务

生成每周销售报告的服务。虽然销售报告很重要，但是生成服务的短暂故障不会造成任何严重影响。

市场电子邮件发送服务

定期向用户发送电子邮件的服务。如果服务宕机一段时间，邮件发送可能会被推迟，但是通常不会对用户造成任何严重的影响。

121

示例：在线商店

图 16-4 是一个由许多服务组成的在线商店示例程序。每个服务都有一个标签来表示被分

配的服务级别。

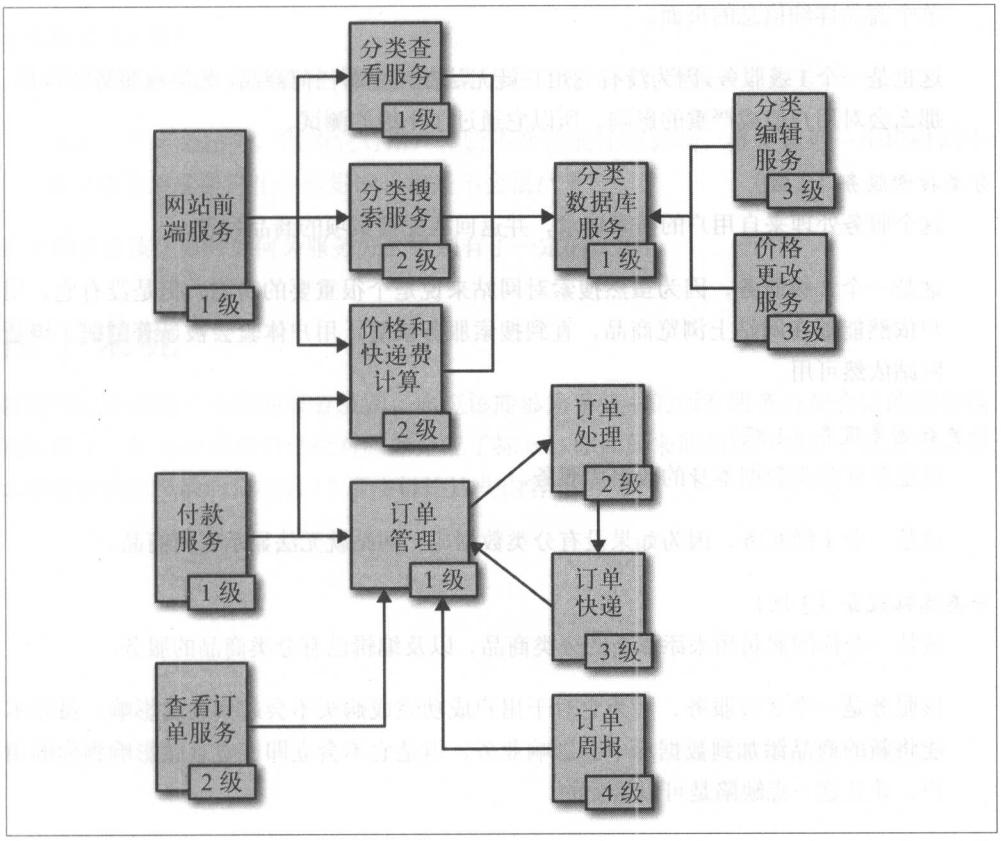


图16-4：示例程序：一个在线商店

看着这幅图，从描述中想象一下每个服务的职责。想象一下当服务出现故障时用户可能或应该面对的体验。服务级别应该与用户体验保持一致。

下面提供了一些该应用中的示例服务。

网站前端服务（1 级）

这是生成和展示网站的服务。它负责生成 HTML 页面，与用户在浏览器中进行最直接的交互。

这是一个 1 级服务，因为如果没有它，用户就无法访问你的整个在线商店。如果该服务不可用，那么会对用户造成严重的影响，所以它通过了 1 级的测试。 122

分类查看服务（1 级）

该服务会读取分类数据库，并将合适的分类数据发送给前端服务。它用来生成展示单个商品详细信息的页面。

这也是一个 1 级服务，因为没有它用户就无法在线查看任何商品。如果该服务不可用，那么会对用户造成严重的影响，所以它通过了 1 级的测试。

分类搜索服务（2 级）

这个服务处理来自用户的搜索请求，并返回匹配搜索项的商品列表。

这是一个 2 级服务，因为虽然搜索对网站来说是个很重要的功能，但是没有它，用户依然能够在网站上浏览商品，直到搜索服务恢复。用户体验会被显著削弱，但是网站依然可用。

分类数据库服务（1 级）

这是存储分类数据本身的数据库服务。

这是一个 1 级服务，因为如果没有分类数据库，网站就无法显示任何商品。

分类编辑服务（3 级）

这是一个你的雇员用来添加新的分类商品，以及编辑已有分类商品的服务。

该服务是一个 3 级服务，因为它对于用户成功完成购买不会起到关键影响。虽然不能将新的商品添加到数据库中会影响业务，但是它不会立即或者直接影响到你的用户，并且这一点缺陷是可以接受的。

付款服务（1 级）

这个服务用来为用户显示付款流程。如果没有该服务，你的用户就无法购买商品。

这是一个 1 级服务，因为它对你的用户（他们不能购买商品）和业务（用户购买不了商品你就无法挣钱）都会造成非常严重的影响。

123 订单快递服务（3 级）

这是打包并快递用户订单（一个很简单的例子）的服务。没有该服务，你的用户就无法收到他们购买的订单。

这看上去可能应该是一个 1 级服务，因为快递订单对于业务来说很关键。但是我们可以这样想一想：如果你一个小时无法快递订单，对用户会造成什么样的影响呢？对你的业务会造成什么影响呢？大多数情况下，它对用户造成的影响非常有限——一个小时的延迟并不会影响用户何时接收到他们的包裹。它可能对你的业务有一些

影响，因为负责打包订单的雇员可能会在一段时间内无法工作。因为它对业务有一些不严重的影响，也不会对用户造成严重的影响，所以 3 级的标签是合适的。

订单周报（4 级）

这是负责收集订单数据并为财务和管理人员生成每周业务报告的服务。

这是一个 4 级服务，因为它对用户体验不会造成任何影响。报告延迟一小段时间生成可能会对业务产生一些影响，但是不会很严重。

这个例子应该让你对如何为服务分配级别有了一定的认识。

接下来呢

既然你已经理解了不同的服务级别，你应该能够为系统中的所有服务分配合适的服务级别标签了。但是如果我们已经对服务分配了标签，你如何来使用这些标签，以及带来什么样的价值呢？我们会在第 17 章中讨论这些内容。

125

第 17 章

使用服务分级

当为所有服务分配服务级别之后，你如何来使用它们呢？下面是几种常见的方式。

期望

服务期望的运行时间是什么？它的可靠性如何？它存在多少问题？它允许多长时间出现一次故障？

响应性

你应当以多快的速度来响应问题？你可以通过哪些途径来解决问题？

依赖

依赖你的服务，以及你所依赖的服务，它们的服务级别分别是多少？它们对你的服务有什么样的影响？

我们分别来看其中的每一点。

期望

服务的期望对于用户来说是很重要的一个部分。服务等级协议（SLA）是管理这些期望的一种方式。由于这部分非常重要，所以我们会在第 18 章用一整章来深入讲解这个话题。

响应性

当系统中发生某个问题时，你对问题的响应性取决于以下两个因素：

126

- 问题的严重性
- 出现问题的服务级别

1 级服务的高严重性问题应该比 3 级服务的高严重性问题更加重视对待。这很显而易见，但是如果 1 级服务发生了一个中严重性的问题，可能需要比 3 级服务的高严重性问题更快地响应。图 17-1 说明了这一点。

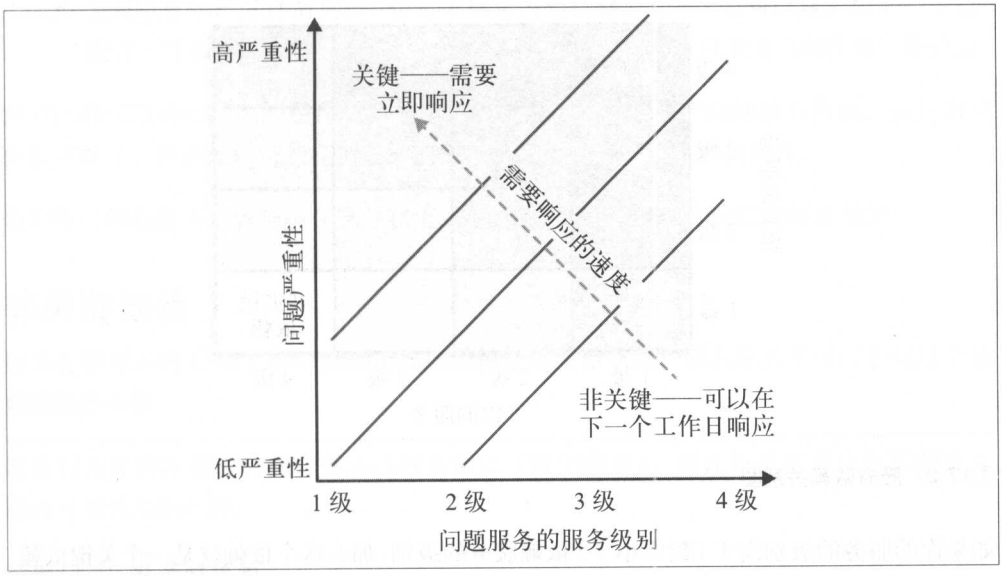


图17-1：不同服务级别和问题严重性需要的响应性

问题越严重，或者服务越重要（服务级别数字更低），那么就需要越及时、越紧急地响应。图 17-1 中的几条平行线表示相似的响应重要性。1 级服务的中低严重性问题，可能需要与 3 级服务最严重的问题具有相似的响应程度。4 级问题几乎永远也不需要紧急响应。

另外，2 级服务的低严重性问题的响应程度，可能与 4 级服务的高严重性问题相似。

你可以借助图中的信息来调整响应性的各个方面。例如，你可以使用响应性级别来决定以下几件事：

- 哪些服务的哪类问题需要立即发送报警通知。
- 期望的 SLA。
- 对于低优先级问题的上报路径。
- 提供响应的时间安排（24 × 7 或者仅办公时间）。
- 是否提供紧急部署或者产品更改。
- 根据服务的可用性和响应性所制订的 SLA。

127

依赖

在构建服务的时候，该服务被分配的级别与其他依赖服务的级别之间的关系非常重要。图 17-2 展示了不同服务级别之间的关系。

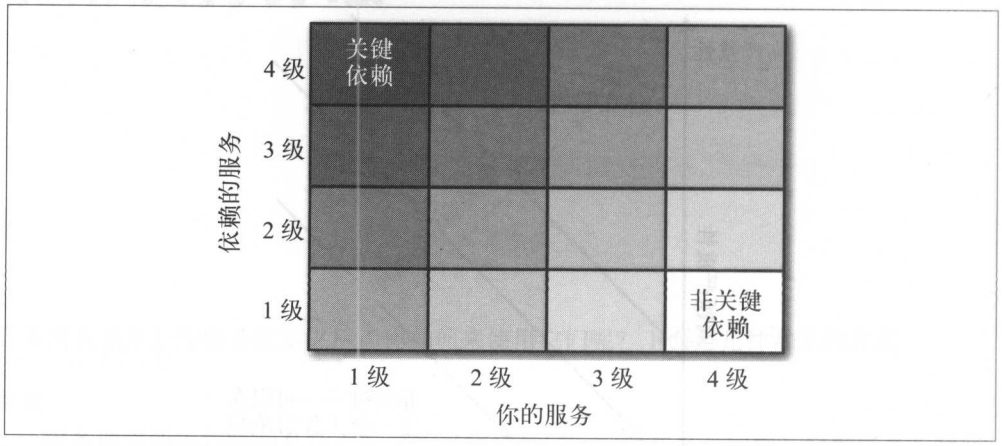


图17-2: 服务依赖关系图

如果你的服务的级别高于（数字小于）依赖服务的级别，那么这个依赖就是一个关键依赖。如果你的服务的级别低于（数字大于）依赖服务的级别，那么这个依赖就是一个非关键依赖。

关键依赖

如果在参考示例 17-1 之后，你确定你的依赖是关键依赖，那么作为一个服务的开发人员，你需要小心处理依赖发生故障的情况，避免对服务造成严重的影响。

128 如果关键依赖出现故障，你的服务应当仍然能够尽力提供功能。因为依赖服务的级别比较低（数字较高），意味着它无法拥有同当前服务一样的可用性和可靠性。

示例17-1: 关键依赖

请你查看一下图 16-4 所示的系统，关注一下 1 级的网站前端服务。当这个服务试图向用户展示某个产品的详细页面时，它需要确定该产品的当前价格。为了做到这一点，它需要调用价格和快递费计算（PSCC）服务来获取商品价格。

如果 PSCC 服务（2 级服务）不能提供服务呢？网站前端服务（1 级服务）仍然必须尽可能地提供功能。那么，这个时候它应该怎么办呢？

它需要优雅地处理来自 PSCC 服务的失败消息（或者响应丢失）。一旦它确定 PSCC 服务发生故障，它需要立刻决定如何来显示产品详情页面。以下是一些备选项：

- 它可以在页面上使用之前缓存的价格（如果之前缓存过的话）。
- 它可以显示产品详情页面，但是不显示当前价格。相反，它可以显示例如“无货”或者“当前价格未定”，或者甚至“添加商品到购物车中以查看当前价格”的消息。

用户仍然可以看到产品的图片、其他用户的评论，以及其他产品的细节内容。虽然用户体验下降了，但是他们仍然能够在你的网站上完成一些非常重要的操作。

我们称这种办法为优雅降级（已经在第 13 章中详细地介绍了如何处理服务故障）。

非关键依赖

如果在参考示例 17-2 之后，你确定你的依赖是非关键依赖，那么你几乎可以忽略这个依赖的服务失败。

这是因为你的依赖服务拥有更高的服务级别（数字较低），因此也会拥有比你的服务更高的可用性和响应性。

示例17-2：非关键依赖

我们再次以图 16-4 所示的在线商店系统为例，但是这次关注于订单周报这个 4 级服务。因为它需要获得生成报表的信息，它需要调用订单管理这个 1 级服务。

129

如果订单管理服务出现失败会发生什么呢？订单周报服务应该如何应对呢？其实订单周报服务也可以直接失败了。因为订单管理服务是一个 1 级服务，它的任何问题都需要非常快速地被处理，因此响应性和紧急性都非常高——远高于对订单周报服务失败的处理。

因此，当订单管理服务出现故障时，根本不需要对订单周报服务做任何特殊处理，因为如果订单管理服务不可用，周报服务不可用也是正常的。

小结

服务级别为服务所有者、依赖服务和用户提供了一种展现服务关键性的便捷方式，从而让它们可以轻松地理解和沟通服务之间的各种期望。这种简易性降低了出错的几率，并且服务级别建立的简单模型，让人们对服务之间的沟通变得更加容易。

131

第 18 章

服务等级协议

期望管理。

这就是服务等级协议的核心内容。如我们在第 17 章所讨论的，每个人对服务的期望都是不同的。许多期望都与服务的服务级别相关，但是当我们更深入了解的话，这些期望会更加具体。

本书中讨论的服务等级协议（SLA），并不是公司和用户之间的法律或合同协议。相反，这是团队和服务所有者之间的协议，它们提供了一个沟通服务间期望的机制。

什么是服务等级协议

服务等级协议是一个提供某种级别可靠性和性能的承诺。

它们用来在服务所有者和用户之间创建一个牢固的合约关系。

例如，一个隔天快递服务的 SLA 可能是在第二天上午 9 点之前发送某个包裹。而一个航线的 SLA 可能是在航班到达后的某段时间内快递包裹。一个电力公司的 SLA 可能是在多短时间内修复风暴之后的电力故障。

示例18-1：什么是SLA

我们以图 16-4 所示的在线商店系统为例。

132 用户希望即使他们不使用网站时，网站依然是运行着的——即他们希望网站是高可用的。他们还希望网站加载速度非常快，这样在使用时就不会有延迟。此外，他们还希望在网站上能买到想要的商品，希望库存充足可以随时邮寄。最后，他们希望在支付订单后，能够在合理的时间内收到他们订购的商品。

这些期望都可以表述成 SLA。例如：

可用性

用户希望网站随时可以访问。你可以将这一点描述成网站可运行的最小时间百分比。例如，一个可用性 SLA 可能会这样描述，“我们的商店至少 99.4% 的时间是可用的”。

加载时间

用户希望页面快速加载——即希望网站看上去是立刻响应的。你有很多种方式来描述这一点，但是最简单的方式是用页面需要加载的最长时间——例如，“99% 的时间页面会在 4 秒内加载”（请参考本章后面的“排名 SLA”小节）。

产品

用户希望在商店中买到他们想买的商品，也期待这些商品有库存并且能够立即发货。你可以将这一点描述成一个百分比，例如，“至少分类中 80% 的商品都有现货”。

快递

用户期望他们购买的产品能够尽快到货。你可以用从下订单到发货的时间，或者以商品到达用户手中的时间来描述这一点。例如，“我们在 24 小时之内送达商品”。

以上所有这些都是 SLA 的例子。虽然它们之间存在很大区别，但是目的都相同，就是描述用户对系统的期待是什么样子的。

你可以在系统运行和与用户交互时，来测量这些 SLA 的实际情况。你可以生成一些图表来展示不同时间的测量结果。但是 SLA 指标是服务正常运行的底线。图 18-1 展示了商品库存的情况，对指定时间拥有库存商品的百分比进行了测量。

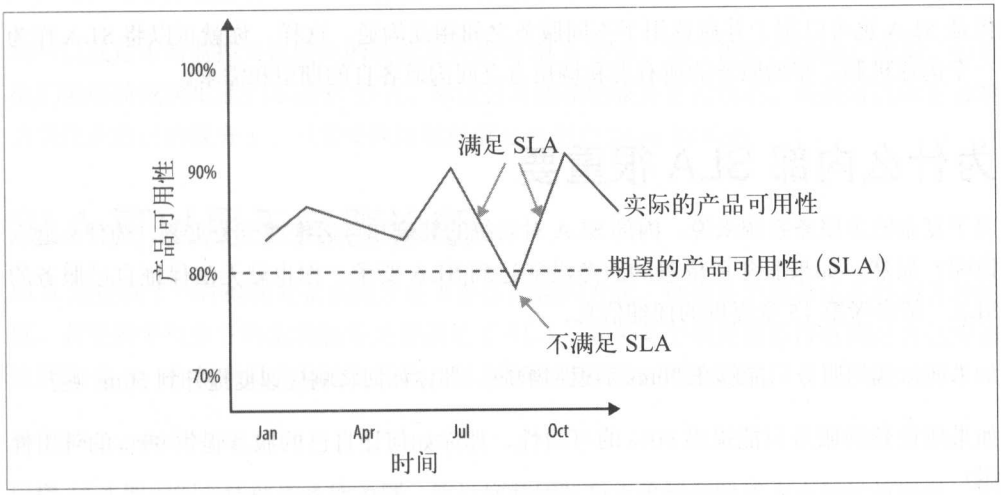


图18-1：SLA的比较情况示例

133 从图中你可以看到，现货商品的百分比会随着时间发生变化。你还可以看一下 SLA 线，它表示你期望的情况是高于 80%。

大多数时候，现货商品的百分比都是高于 SLA 的（我们称为满足 SLA）。但是，夏末的某段时间内，它下降到了 80% 的 SLA 以下（我们称为没有满足 SLA）。

在某些行业中，用户会与公司签订条约，要求满足一定的 SLA 要求，如果没有满足，可能会有一些赔偿或者其他规则。

例如，Amazon 的 AWS 就与用户签订了 SLA，如果它没能满足这些 SLA，会提供一定的经济补偿。

以 Amazon EC2 为例，如果 AWS 的月运行时间低于 99.95%，它会给受影响的用户提供 10% 的费用减免。如果低于 99.0%，它会给予 30% 的费用减免。你可以在 *https://aws.amazon.com/ec2/sla/* 了解到 AWS 如何计算 SLA 和费用的详细信息。

通过 SLA 来监控系统的情况，对于内部业务使用来说非常有帮助（确保系统能按照用户
134 预期的标准来执行）。或者，像 AWS 一样，SLA 可以为用户提供财产方面的承诺。不管怎样，两种情况下的 SLA 和测量 SLA 的方法都是一样的。

外部 SLA 与内部 SLA 的对比

在示例 18-1 和 AWS 这两个案例中，主要介绍了外部的 SLA。为了能够描述用户使用系统的情况，我们需要对这些 SLA 制订标准和进行监控。

但是 SLA 也可以用于并应该用于不同服务之间相互沟通。这样，你就可以将 SLA 作为一个沟通机制，帮助服务的所有者和调用方之间沟通各自的期望和需求。

为什么内部 SLA 很重要

对于复杂的多服务系统来说，内部 SLA 对系统的健康和可运维性至关重要。为什么这么说呢？显然，如果服务所依赖的服务没能达到 SLA 要求，你也就无法保证自己服务的 SLA。请参考第 15 章提供的详细信息。

如果所依赖的服务只能够在 90ms 后返回响应，那你如何将响应速度提升到 50ms 呢？

如果所依赖的服务只能提供 90% 的可用性，那你如何让自己的服务提供 99% 的可用性呢？

SLA 可以作为一种信任的手段

SLA 的意义在于如何用高度分布和可扩展的方式来建立服务之间的信任。当你相信某个依赖服务可以达到它的期望值，你就有信心来设置自己服务的期望值。

示例18-2：建立信任

以图 16-4 所示的在线商店系统为例。假设你和团队负责价格和快递成本的计算服务。你的用户是网站的前端服务和结算服务。它们主要依赖于你的查找指定商品价格的功能。由于这些服务都使用该值来生成展现给最终用户的网页，所以它们需要迅速查找到相应的价格。你的团队同意保证每个查询价格的请求在 20ms 内返回。

现在，当你要满足这个承诺时，你意识到服务还需要快速访问分类存储服务，其中包含你需要计算价格的数据。但是，考虑到你已经给了 20ms 的承诺，你会担心分类存储服务是否能足够快速地提供所需数据，而分类存储服务是由另一个团队负责的。你如何能确定那个团队有能力满足你的性能需求呢？你有两个选择。

第一个选择是联系分类存储服务团队，深入了解它们服务的工作原理，找到性能问题。然后，分析这个团队是否能够满足你需要的性能。当然，这种方式是高度侵入式的，成本昂贵并且在大型组织中不太可能实现。

另一个选择是与分类存储团队进行协商，在服务的性能 SLA 上达成一致。假设你通过与团队的沟通，它们同意每个响应小于 10ms。你知道如果他们能够如此迅速地响应，你就可以达到对用户承诺的 20ms。

只要他们能够保证自己的 SLA，你也可以保证你的 SLA。

你可以通过不断监控其他团队的性能是否达到 SLA 要求，来了解他们的运行情况。如果他们能够持续满足他们承诺的 SLA，你就会对依赖的服务更有信心，从而可以将更多精力关注在自己的服务上，只需要保证继续满足对用户 20ms 的承诺。

SLA 可以用于问题诊断

SLA 还提供了一种检测复杂系统中是否存在问题的方式。如果某个服务正在经历各种问题，首先就要检查它的依赖服务是否满足了 SLA。如果某个依赖服务没有满足自己应该达到的 SLA，你就可以先从这点开始，逐步诊断服务发生问题原因。

示例18-3：查找问题

以图 16-4 所示的在线商店系统为例。假设你和团队负责价格和快递成本的计算服务，如

示例 18-2 中所述。

现在，假设你某天半夜接到一个电话。你的服务在提供价格查询功能时变得缓慢，并且影响到了公司的用户。你开始检查服务性能是否能够达到 20ms 以内，但是发现平均每个查询请求要 500ms 才能返回。这肯定会影响到前台页面的响应速度，从而令用户感到不满。

但是，是什么导致的问题？是服务中出现了什么问题吗？还是其中一个依赖服务存在问题？

136 可能是你的服务出现了某些问题——可能是硬件或者其他方面。但是，在你打算花费大量时间试图去寻找服务的问题之前，最好先检查一下依赖服务的性能。

由于我们知道服务依赖于分类存储服务，并且与它们之间达成的 SLA 约定是 10ms，所以我们检查该服务性能后发现它也存在着性能问题。调用该服务的请求不仅不能在 10ms 内返回，甚至超过了 400ms。显然，该团队正在遭遇某个性能问题。你找到并拨打了他们的值班电话，发现他们正在处理这个问题。

当意识到这很可能导致了你的性能问题，你开始跟踪其他团队解决问题的进展。这比花费大量宝贵时间寻找自己服务的问题更有意义。

通过与依赖服务建立定义良好的 SLA，当你自己的服务或者依赖服务出现问题时，你可以非常容易地定位问题发生原因。

SLA 的性能检测方法

有许多用来检测服务性能的方法，但具体使用哪种方法，要根据服务的用户和所有人的需求来决定。以下是一些常见的性能检测方法。

调用延迟

这种方法用来测量服务需要多长时间来处理一个请求并返回响应。通常测量结果以微秒或者毫秒计算。服务的消费者必须要知道一个请求的处理时间，因为这部分时间会计算在处理请求的总时间内。这就是示例 18-2 和 18-3 中使用的 SLA 类型。

流量

这种方法用来测量服务在一段时间内能够处理多少请求。通常按照每秒请求数量来测量，服务的所有人必须知道来自消费方服务的流量，才能知道自己的服务是否能够满足对方的期望。

运行时长

这种方法用来测量服务正常、无故障运行的时间。通常以百分比计算，它可以测量服务在一段时间内（通常以天、月或者年计）的可用性。

错误率

它用来测量服务在一段时间内出现过多少次失败。通常以百分比计算，通过一段时间内失败的请求数除以请求总数得出结果。

限定 SLA

限定 *SLA* 通常指期望满足的一个限定值。如果实际的性能好于限定值，则说明我们满足了SLA。如果实际的性能差于限定值，说明我们没有满足SLA。限定值本身就是SLA的值。

例如，“调用速率必须 < 1000 请求 / 秒”表示对服务预期流量的一个限定 SLA。

另一个例子，“请求响应必须 < 20 ms”表示对服务调用延迟的一个限定 SLA。

你可以在大多数性能检测中使用限定 SLA。

排名 SLA

当你可以测量某个值，并且可以确保该值始终好于限定值时，适合使用限定 SLA。这种 SLA 更适合于描述可用性、运行时间和错误率。

另一种 SLA 测量方法是*排名 SLA*。当某个事件的实际性能差异较大时，你可以用它来测量这个事件的性能。

排名 SLA 适合测量调用延迟等值。通常，服务处理每个请求的时间可能会差异较大，大多数情况下我们并不关心每个请求的处理时间，只要绝大多数请求可以在限定时间内处理就好。

排名 SLA 用一个高于或低于某个指定值的百分比数值来表示，通常的形式如下：

```
TP< 百分比 > 低于 < 值 >
```

另一个例子：

```
TP90 低于 20 毫秒
```

我们可以将其读作“90% 的请求在 20 毫秒以内处理”。

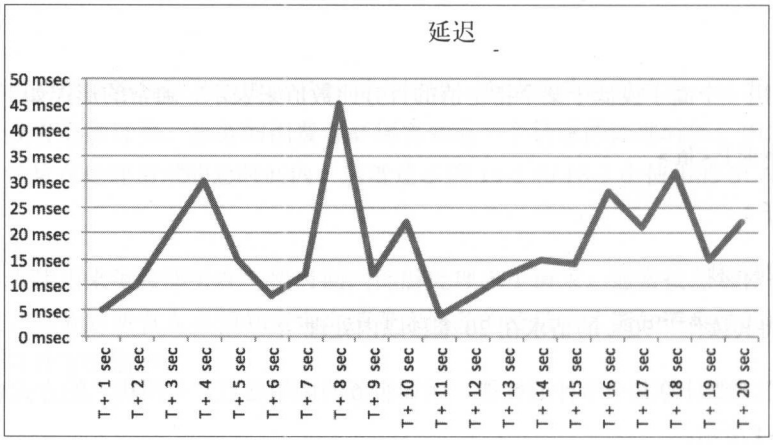
通常，我们需要计算某个事件的性能，例如服务的调用延迟，并用排名的方式表示出来，如示例 18-4 所示。

138 示例18-4：用排名方式来表示的调用延迟

假设我们有一个可以响应外部请求的服务。在经过一段时间的观察后，我们记录下了这些调用的延迟时间：

Service Call Latency - Actual	
Req Time	Latency
T + 1 sec	5 msec
T + 2 sec	10 msec
T + 3 sec	20 msec
T + 4 sec	30 msec
T + 5 sec	15 msec
T + 6 sec	8 msec
T + 7 sec	12 msec
T + 8 sec	45 msec
T + 9 sec	12 msec
T + 10 sec	22 msec
T + 11 sec	4 msec
T + 12 sec	8 msec
T + 13 sec	12 msec
T + 14 sec	15 msec
T + 15 sec	14 msec
T + 16 sec	28 msec
T + 17 sec	21 msec
T + 18 sec	32 msec
T + 19 sec	15 msec
T + 20 sec	22 msec

可以根据这些值绘制一幅如下所示的图表：



通过该数据，我们可以计算出该服务的一些最高延迟。

TP90

这表示 90% 的延迟都低于该值。在本例中，90% 的数据就是 18 个数据点。除去最高的两个数据点（45 毫秒和 32 毫秒），只剩下 18 个数据点，其中最高值是 30 毫秒。因此我们说 TP90 是 30 毫秒。

TP80

这表示 80% 的延迟都低于该值。在本例中，这意味着去掉 20% 的最高值（4 个数据点：45、32、30 和 28 毫秒）。在剩下的 16 个数据点中，最高的值是 22 毫秒。因此我们说 TP80 是 22 毫秒。

以此类推，以下是一些可以从数据中计算出的其他 TP 值：

```
TP95 = 32 msec
TP90 = 30 msec
TP80 = 22 msec
TP50 = 14 msec
```

以下是其他一些可能会用到的值：

```
TPmax = 45 msec（最大值）
TPmin = 4 msec（最小值）
TPavg = 18 msec（平均值）
```

这个排名显然会随着时间发生变化。当你计算出这些值之后，可以使用限定 SLA 来定义期望值。例如，在示例 18-4 中，你的服务的 SLA 可能如下：

```
TP90 < 35ms
```

如果做到了这一点，那么服务就满足了自己的 SLA。但是，如果它之前承诺的是如下 SLA：

```
TP80 < 20ms
```

那么服务就无法满足自己的 SLA（当前 TP80 是 22 毫秒）。

延迟分组

有些时候，SLA 可以按照相关性进行分组。例如，某个服务可以保证一定的延迟，但是必须是在合理调用量的前提下。因此，一个 SLA 可能有如下表示：

```
当流量 < 250k 请求 / 秒时，调用延迟 TP90 < 25 毫秒
当流量 > 250k 请求 / 秒并且 < 400k 请求 / 秒时，调用延迟 TP90 < 30 毫秒
```

140 究竟应当定义多少内部 SLA，以及定义哪些内部 SLA

你在创建服务时可能会想问，究竟应该为服务定义多少个内部 SLA？

首先，应当保证尽可能少的 SLA 数量。当 SLA 的数量增加时，SLA 的意义和相互影响会变得非常复杂。

你应当确保 SLA 覆盖了服务的所有关键部分，为所有的主要功能都定义合适的 SLA，尤其是对业务至关重要的地方。

你应当与服务的消费方一起来协商 SLA，因为不能满足消费方需求的 SLA 就是一个无用的 SLA。但是，尽量对所有的消费方都使用一样的 SLA。服务应当尽可能用一套 SLA 来满足所有消费方的需求，因为为每个消费方都建立一组 SLA，不仅会极大增加复杂性，也不会带来任何实际的价值。

你应当只定义那些实际中可以实现监控和报警的 SLA。如果你无法有效地验证 SLA，定义它也没有任何意义。此外，应当关心服务是否有违反 SLA，因为这应该是问题发生的最先表现，因此你需要确保当服务不满足 SLA 时可以第一时间接收到相关报警。

你可以对超出 SLA 的值进行监控和报警，并将在它们之上的值作为内部的 SLA。这些数据在查找和管理服务问题的时候非常有用，同时又不会实际承诺给消费方。

你应当建立一个包含所有 SLA 和监控的仪表盘界面，这样一眼就可以发现正在发生的问题。并且你应当将这个界面共享给所有的依赖方，这样他们也可以看到你的服务情况。

除此之外，你需要确保可以访问到所有依赖服务的仪表盘界面，这样你可以监控它们是否正在发生问题，以确定问题是否会影响到你的服务。

141 关于 SLA 的其他评价

监控和使用 SLA 会很快得到广泛应用，并且你可以很容易快速了解到 SLA 的各项监控细节。

理想情况下，我们的目标不只是建立对所有 SLA 的监控，而是发掘一个可以用来对比的数值。任何数字都好过没有数字。内部 SLA 的目的不是为了增加数字，而是为当前服务和其他依赖服务提供指导，帮助团队之间设置合理的期望。

SLA 可以并且应该成为你与其他团队进行沟通的语言的一部分。

第 19 章

持续改进

如果你的应用程序运行得不错，就会逐渐需要扩大规模以处理更大的流量。处理新增流量需要一些更复杂的机制，同时新增的流量压力也会降低系统的质量。

通常来说，应用程序的开发人员不会从一开始构建系统时就考虑可伸缩性。我们一般会认为系统已经具有足够的伸缩能力，但是结果多半是，系统中的各种问题使得我们无法承载更大的流量和数据量。

当我们开始遇到瓶颈的时候，如何提升系统的可伸缩性呢？显然，我们在系统的生命周期中越早考虑可伸缩性，就越容易做到这一点。但是其实不管在生命周期的哪个阶段，我们都有很多技巧可以提高系统的可伸缩性。

本章会讨论其中的一些技巧。

定期检查你的应用程序

第 I 部分和第 II 部分详细介绍了如何维护一个高可用的应用程序，以及如何管理其中的风险。在你考虑利用技术手段扩展应用程序之前，必须先建立应用程序的可用性和风险管理模型，它们是你必须首先考虑的事情。如果你没有建立起可用性和风险管理模型，随着系统的扩展，你会逐渐无法掌握系统的运行状况，并且系统会开始出现随机的、非预期的各种问题。这些问题不仅会导致系统故障和数据丢失，也会严重影响你构建和提升系统的能力。此外，随着流量和数据量的增加，这些问题只会变得越来越严重。在开始做任何事情之前，一定请先做好可用性和风险管理。

144

微服务

在第Ⅲ部分中，我们讨论了基于服务和微服务的架构。虽然你需要做很多架构方面的决策，以及调整架构的方向，但是应当尽早从单体或者多单体的架构转向面向服务的架构。

服务所有权

当你转向基于服务的架构时，同时也应该转向分布式的所有权模型，由每个独立的开发团队来完全负责他们的服务。这种分布式的所有权架构会提高应用程序在代码复杂性、流量以及数据量方面的扩展能力。我们在第 15 章详细讨论了有关所有权的内容。

无状态服务

当你在构建应用程序，或者将应用程序迁移到基于服务的架构时，要注意在系统中存储数据和状态的方法。

无状态服务指的是自己不管理任何数据和状态的服务。所有服务用到的状态和数据都来自于请求传递过来的值。

无状态服务提供了优秀的伸缩能力。因为它们都是无状态的，所以无论是垂直扩展还是水平扩展，通过添加额外服务器来提升处理能力都是一件很简单的事情。如果你的服务不需要维护状态，就能够在如何扩展和何时扩展服务规模方面获得最大的灵活性。

此外，如果不需要考虑服务状态，你还可以在服务的前端使用某些缓存技术，利用更少的资源来处理更大的流量压力。

数据在哪里

如我们之前所述，当你希望存储数据的时候，显然应当将数据存储在尽可能少的几个服务和系统中，同时避免直接在服务之间暴露数据，减少服务之间对内部数据的了解。

但是，真相却往往并不是如此。

145 相反，我们应当尽可能地将数据保存在本地。服务必须使用的数据应当存储在服务本地，而其他数据应当存储到其他服务器或者数据库中，离需要这些数据的服务越近越好。

将数据保存在本地能带来以下几点好处。

减少单个数据集合的大小

由于你的数据分散在多个数据集合中，每个数据集合的体量就会更小。越小的数据

体积意味着数据交互越少，这使得扩展数据库变得更加容易。这种方式也被称为功能性分区，即将数据根据不同的功能进行拆分，而不是基于数据的体量。

本地访问

通常，当你访问数据库或者存储器中的数据时，都是在访问一条记录或者一段记录。很多时候，其实你并不一定需要其中的很多数据。通过将数据分散到多个数据集合中，也减少了不必要的数据查询量。

优化访问方式

通过将数据分散到不同数据集合中，你可以对每个数据集合进行适当的优化。例如，某个数据集合是否应当保存到关系型数据库中？还是应当保存到键 / 值数据库中？

数据分区

数据分区的含义有很多种。在本节中，它表示将数据根据其中的某些键划分到不同的数据段中，通常为了利用多个数据库来存储更大规模的数据集合，或者获得比单台数据库更高的访问速度。

数据分区还有其他的类型（例如之前提到的功能性分区），但是，在本节中，我们只关注基于分区键的数据分区。

举一个简单的数据分区示例，即将某个应用程序的所有数据按照账号进行分区，这样所有账户名称由 A-D 开头的数据被划分到一个数据库中，所有账户名称由 E-K 的数据被划分到另一个数据库中，以此类推（如图 19-1 所示）。[注1] 这是一个非常简单的例子，但是应用程序的开发人员通常会采用数据分区技术，来大幅提升可以访问应用程序的用户并发数量，同时支持更大规模的数据量。

一般来说，你应当尽可能避免使用数据分区。为什么？因为一旦像这样对数据进行分区，就可能遇到以下这些问题：

146

- 增加了应用程序的复杂性。因为从现在开始，你在实际获取数据之前，需要先去计算数据存放在哪里。
- 无法很容易地获取多个分区中的数据。这一点在做业务数据分析查询时非常有用。
- 需要很谨慎地选择建立分区所依赖的分区键。如果你选择了错误的键，可能就会导致数据库分区使用不均衡，某些分区使用频繁而其他很少被使用，这样不仅会降低分区的有效性，同时也会增加数据库管理和维护的复杂性。如图 19-2 所示。

注 1　实际中会经常使用的基于账户的分区机制，是使用账户标识来代替账户名称。本例中使用账户名称是为了方便读者理解。

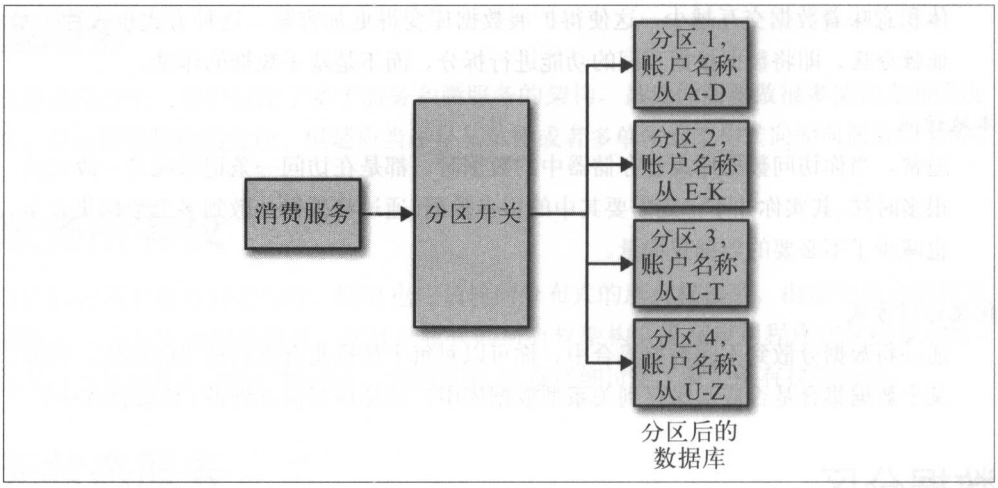


图19-1：根据账户名称进行数据分区的示例

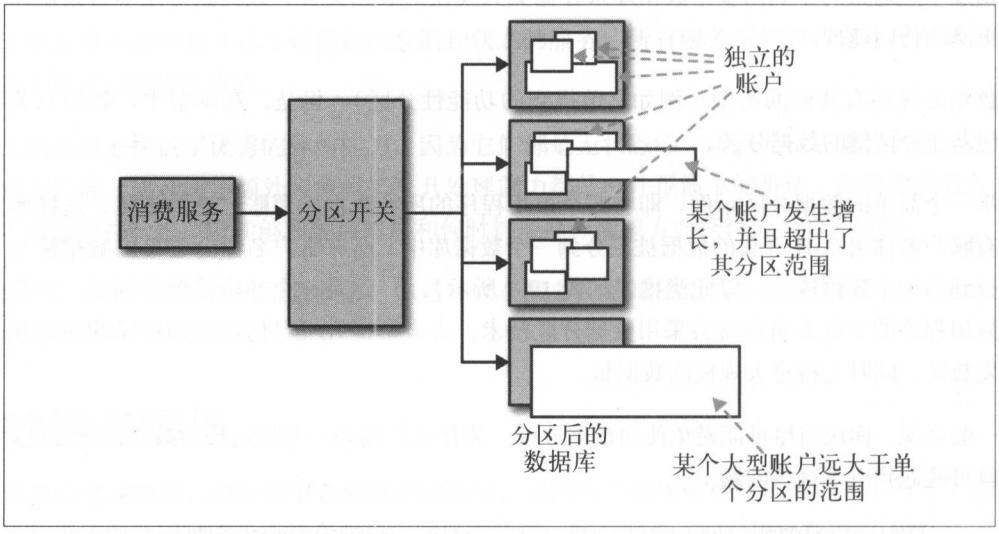


图19-2：账户超出数据分区能力的示例

- 需要偶尔进行重分区来平衡分区之间的数据。根据选择的分区键和数据集合的类型、体量不同，重分区可能变得非常困难、非常危险（数据迁移），甚至在某些时候，几乎无法实现。

一般来说，账户名和账户 ID 都不是好的分区键（虽然它们都最经常被作为分区键使用）。这是因为一个账户的数据体量可能会发生变化。它开始可能很小，很适合分配在一个存

放大量小账号的分区中。但是，如果随着时间它变得越来越大，很快会因为单个分区无法承载压力，所以你不得不重新进行分区，来更好地平衡账户数据。如果一个账号变得太大，甚至超过单个分区的容量，就会导致整个分区方案失败，即使是重分区也无法解决这个问题。图 19-2 展示了这一点。 147

更好的分区键应该是大小尽可能保持固定的键。分区数量增长应当尽可能保持独立和一致，如图 19-3 所示。如果需要重分区，应该是所有的分区都一致地增长，因为分区数据量太大以至于无法处理所以才重新分配分区。

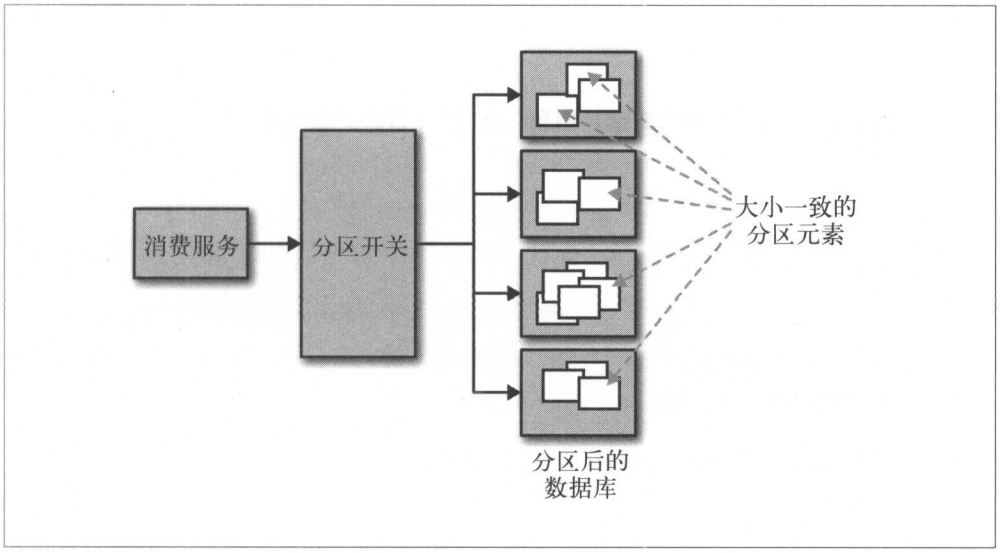


图19-3: 数据保持一致增长的分区示例

实际中可能会用到的分区方案是，选择一个会产生大量小元素的分区键，然后将这些小分区映射到一个更大的分区数据库中。以后如果需要重分区，你只需要更新映射关系，将个别小元素移动到新的分区中即可，这样就避免了对整个系统的大范围重分区。[注2] 148

持续改进的重要性

如今大多数应用程序都经历着流量、体量、复杂性，以及开发维护人员数量的增长。

通常，我们会忽视这些增长所带来的问题，直到我们遇到瓶颈不得不开始解决它。但是，

注 2　如何选择合适的分区键本身就是一门艺术，也是许多书籍和文章所关注的焦点。

那个时候通常已经晚了。各种问题已经积压到很严重的程度，很多早期能够解决问题的简单技巧，到这个时候已经不再适用了。

如果能够在到达这种程度之前及时思考如何保持系统的长期增长，就可以预防很多问题的出现，不断改进应用程序，使它们安全平稳地度过增长期。

第Ⅴ部分

云服务

今日天气预告：“多云，范围可能变广……”

151

第20章

变化和云服务

云服务已经改变了我们构建和运行应用程序的一些理念。但是，当我们还在考虑如何使用云服务来改变应用程序的时候，云已经发生了变化，并且同时改变了我们对云的理解。

云服务有哪些变化

云服务在过去的十年间已经逐渐变得成熟。云服务厂商也增加了很多类型的产品，他们不再只是简单提供诸如文件存储、计算能力等资源。像 AWS 就提供了 50 种不同的服务，来满足各式各样不同的计算需求。

因此，对于我们和应用程序来说，云服务带给我们最大的变化是什么？下面的章节揭示了其中的一些关键点。

对基于微服务架构的认可

正如我们在本书中讨论的，基于服务或者微服务的架构在最近几年已经变得越来越普及，为了减少技术债务、让应用程序更易维护，将应用迁移到基于服务的架构正在成为一种标准技术。

随着各个公司将他们的应用迁移到云服务上，云服务已经成为他们整体产品升级策略的一部分。这些策略中还包含向最先进的应用程序架构迁移，例如这几年趋之若鹜的微服务架构和其他服务化架构。也正是因为像 Docker 这些技术的出现，使得应用程序开发使用微服务架构成为可能。

152 意识到这一点，云服务厂商已经开始提供一些价值更高的产品，例如用来管理微服务容器的 EC2 容器服务。

更小、更专业的服务

随着将应用程序逐渐迁移到云上，我们开始思考如何将云服务用到自己的服务中。一些过去由应用程序自身提供的功能，目前都已经被云服务所代替。

如今一些主流的云服务厂商，已经可以提供诸如数据库、缓存服务、队列服务、日志服务、CDN 以及音视频转码等多项服务。

更专注于应用程序

云服务已经让我们不必再过多关注于基础设施的创建和管理，能够将更多精力用在应用程序和相关环境上。此外，云服务也很大程度地降低了应用程序的管理负担，使得我们可以更专注于解决程序如何运行的问题。

微型初创公司

云服务使得一些非常小的初创公司也可使用相关服务。他们经常是自筹资金、个人运营，但通过云服务，他们也能够使用一些廉价的、可扩展的计算服务或其他服务。

对于一个有想法的人来说，将其想法实现并从中赢利并不容易。如果不需要昂贵的基础设施投入，就可以快速建立一个计算生态系统，无疑能够帮助他们将一些新奇的想法快速投向市场。尤其是一些在线游戏的移动应用，它们从这种能力中收获颇丰。

这些初创公司通过最少的投入让产品快速上线，去市场中检验成功与否。对于那些蓬勃发展的公司，云服务为其应用提供了廉价且易于扩展的能力，使得公司可以将基础设施的投入，维持在与业务发展合理的比例上。从财务的角度讲，这让初创公司更加容易运营和管理。

安全和合规已经成熟

在云服务出现的早期，安全问题是很多公司拒绝使用云服务的一个主要因素。

意识到提升安全的需求后，云服务厂商为云应用提供了更好的安全能力，也以 PCI、SOC 和 HIPPA 等安全法规的形式加强了安全保障。 153

通过不断增加高等级的安全措施，安全因素已经不再是公司迁移到云服务的障碍。

变化还在继续

变化是不可避免的，云服务已经改变了我们构建和运行应用程序的观念。我们已经开始

构建一些更小的、更专业的服务，并且已经学会了如何处理日益增长的海量数据，也将更多的精力从基础设施建设，转移到了应用程序本身。小公司的模式变得越来越可行，同时带来了更多新的想法和创意。安全也已经成为我们做每件事的标准。

云服务已经成熟，我们也逐渐越来越熟练地使用云服务。这在未来会继续发展，我们必须不断适应并紧跟未来的变革。只有那时，我们的应用程序才能持续保持成长和发展。

无论好、坏或者其他，云服务已经发生了变化，并且会改变我们所有人。

第 21 章 <155>

云上的分布

我们都知道将应用分布在多个数据中心的好处，同样的原则也适用于云服务。当将部分或者整个应用部署到云服务上时，我们需要去了解它们在云服务中所处的位置。应用程序如何在云服务中分布，与它们在数据中心中如何分布同样重要，这一点对于应用的伸缩性尤其重要。

但是，云服务让你更难了解应用程序如何分布，也让你更难以主动地将应用部署得更加分散。一些云服务厂商甚至没有提供足够的信息，让你能够知道应用程序运行的地理位置。

幸运的是，像 AWS 这种大型云计算厂商，虽然不会告诉你程序运行的具体位置，但是会提供足够的信息让你来决定程序在哪里运行。

如果你想理解这些信息并加以利用的话，需要先理解 AWS 的架构设计。

AWS 的架构

首先让我们讨论一些 AWS 中的术语。

AWS 区域

云资源相连形成的一大片地区称为 AWS 区域，表示一个特定的地理区域。一般来说，一个区域指的就是一个大洲或者国家的一部分（比如西欧、亚太东北地区或者美国西部）。

它们描述并记录了云资源的地理多样性，并且由多个可用区（AZ）组成，不过，一个区域也可能只有一个可用区。

AWS 区域通常由一个字符串来表示它所在的地理位置，表 21-1 列出了目前的 AWS 区域， <156>

以及它们的名称和所在位置。

表21-1：AWS区域列表

区域名称[a]	覆盖的地理区域
us-east-1	US East Coast (N. Virginia)
us-west-1	US West Coast (N. California)
us-west-2	US West Coast (Oregon)
eu-west-1	EU (Ireland)
eu-central-1	EU (Frankfurt)
ap-northeast-1	Asia Pacific (Tokyo)
ap-northeast-2	Asia Pacific (Seoul)
ap-southeast-1	Asia Pacific (Singapore)
ap-southeast-2	Asia Pacific (Sydney)
sa-east-1	South America (Sao Paulo)

[a] 截至 2016.2 的 AWS 区域和可用区。

AWS 可用区

AWS 可用区是 AWS 区域的子集，表示一个区域指定部分的云资源，但是它们在网络拓扑结构上是彼此隔离的。AWS 可用区描述和记录了云资源在网络拓扑上的多样性。如果两个云资源属于不同的可用区，即使它们属于同一个 AWS 区域，你仍然可以认为它们属于两个不同的数据中心。如果两个云资源属于同一个可用区，它们可能会位于同一个数据中心、同一楼层、同一机架，甚至同一个物理服务器上。

AWS 可用区用一个字符串来命名，以该可用区所在的 AWS 区域名称开始，随后是一个字母（a-z）。表 21-2 列举了一些可用区及其所在的 AWS 区域。

表21-2：AWS 可用区名称

区域名称	可用区名称
us-east-1	us-east-1a us-east-1b us-east-1c us-east-1d us-east-1e
us-west-1	us-west-1a us-west-1b
us-west-2	us-west-2a us-west-2b

数据中心

这个术语并不在 AWS 的术语表中，但是为了能够将传统的非云术语与 AWS 术语对应起来，我们在这里仍然使用它。

157 数据中心是一个用来放置系统资源（例如服务器）的指定楼层、建筑物或者建筑群。

总体架构概述

图 21-1 从整体上展示了 AWS 云服务的架构。AWS 由多个 AWS 区域组成，为了给全世界绝大多数地方提供高质量的访问，这些 AWS 区域地理上分布于全球的各个地方。每个 AWS 区域不仅接入了互联网，并且区域之间可以彼此互通，但是同其他互联网一样，它们也需要使用远程网络连接。

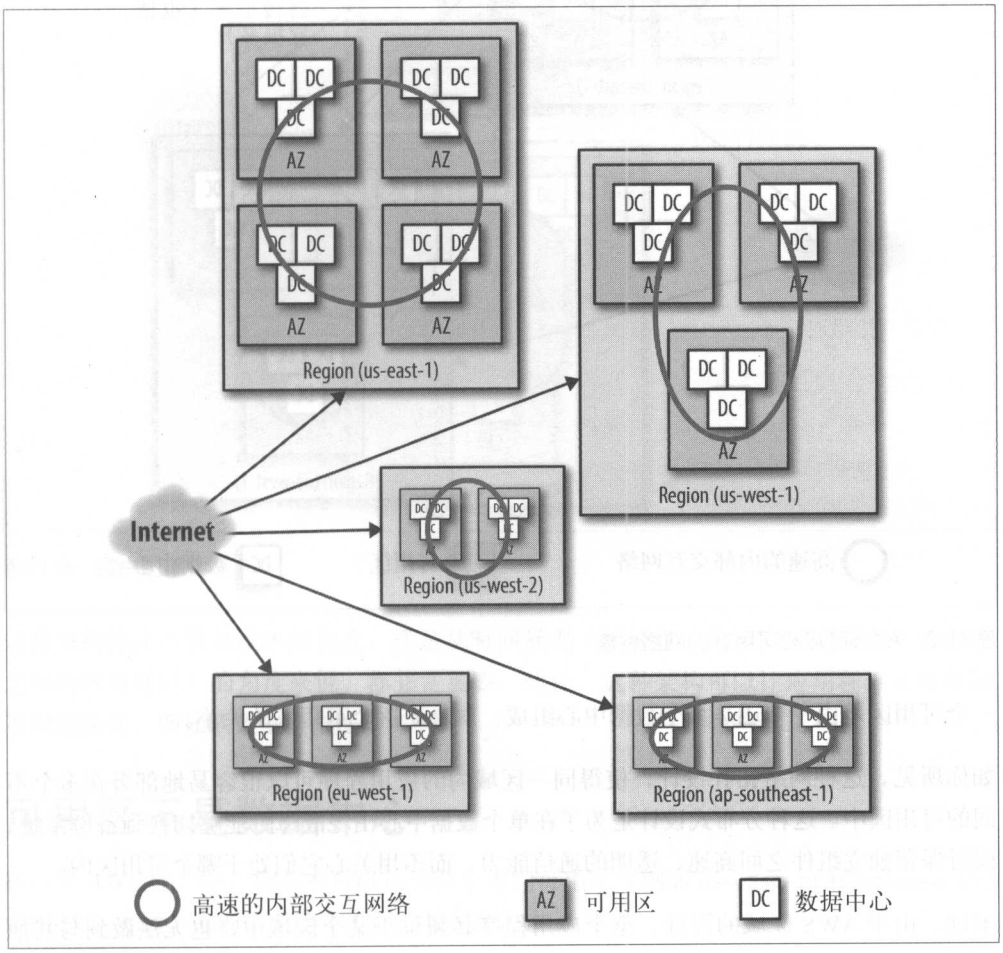


图21-1：AWS数据中心架构

一个 AWS 区域由一个或多个 AWS 可用区组成。如图 21-2 所示，同一个 AWS 区内的可用区之间，通过高速的交换机网络互相连通，这样访问同一 AWS 区域内任意两台服务器的速度几乎一样，而不用去担心它们所处的可用区的位置。 <158>

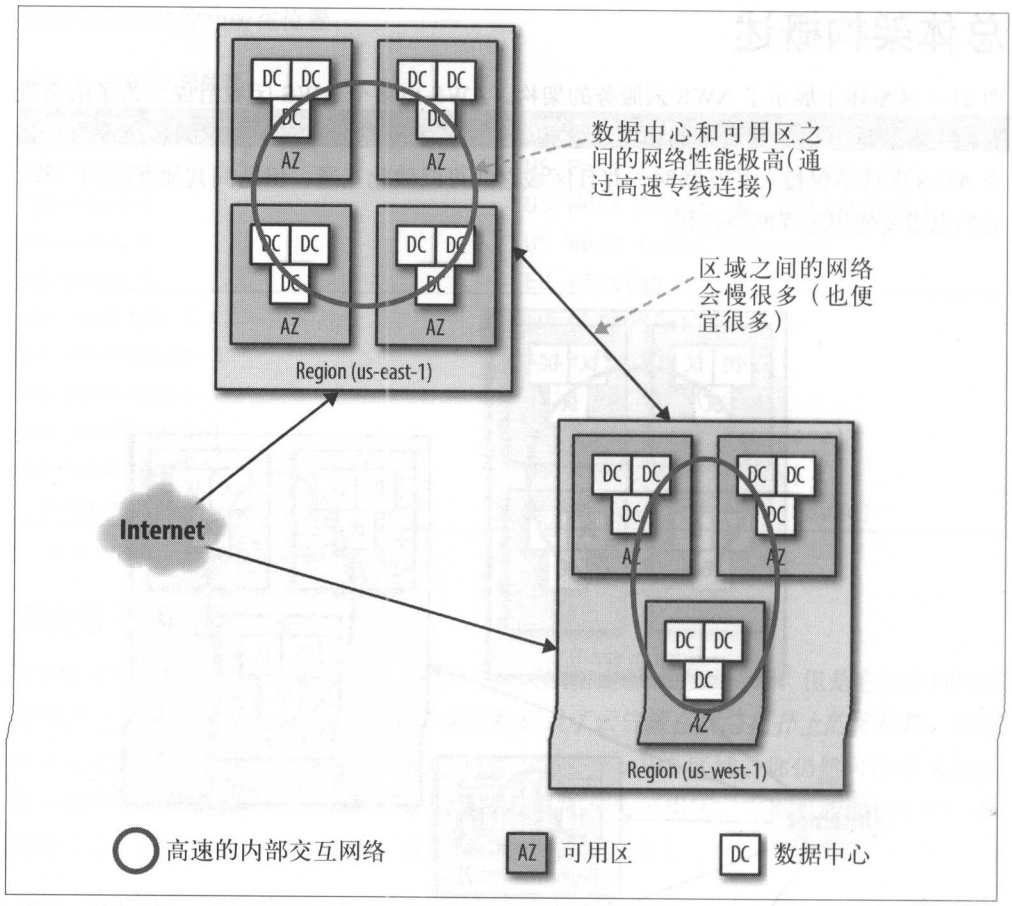


图21-2: AWS区域和可用区的网络性能

一个可用区是由一个还是多个数据中心组成，取决于这个可用区的大小。

如你所见，这种网络拓扑设计，使得同一区域内的应用程序可以很容易地部署在多个不同的可用区中。这种分布式设计是为了在单个数据中心出现故障时能够切换到备份系统，
159 同时保留独立组件之间高速、透明的通信能力，而不用关心它们处于哪个可用区内。

不过，由于AWS区域的设计，整个应用程序必须位于某个区域中，也无法做到与其他区域的高速通信。如果你想在多个区域中运行某个应用程序，那么每个区域都需要能够独立运行一套该程序。这样，每个地理区域都可以在本地访问到应用程序的实例，避免了远程网络通信的代价，如图21-3所示。这也是由AWS网络流量成本模型所决定的，它规定同一区域内的两个可用区之间的通信是免费的，但是跨区域通信或者从区域内访问互联网是需要收取一定费用的。

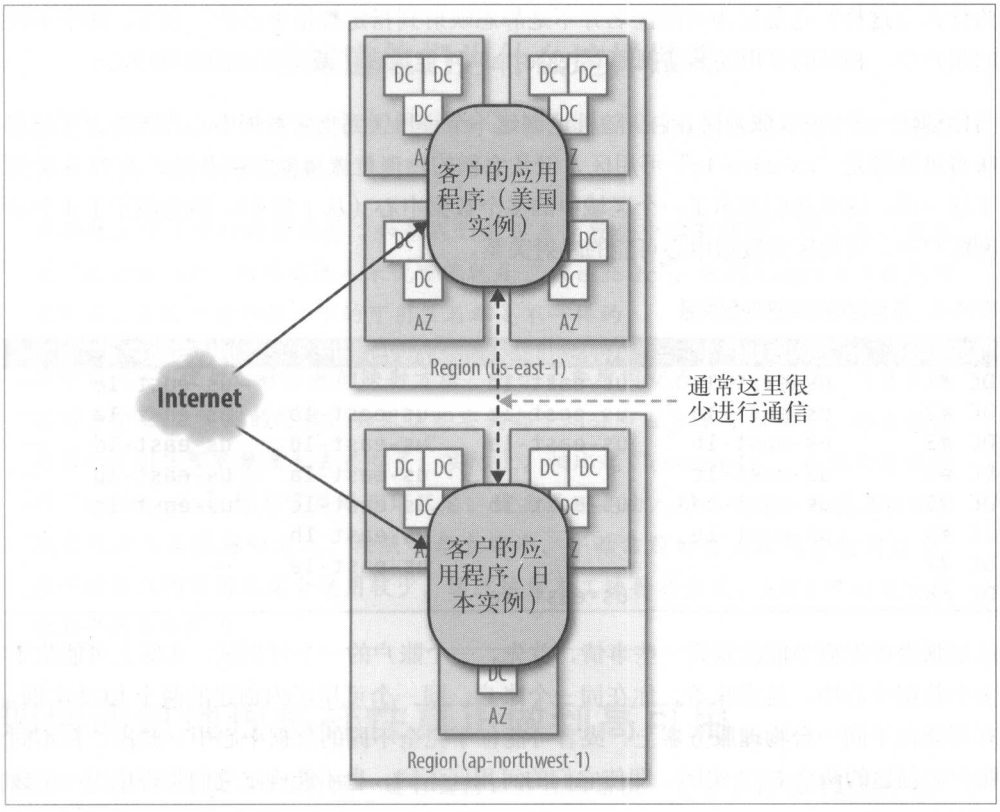


图21-3：客户的架构

这种架构模式不管从成本的角度，还是从时间延迟（区域之间的网络延迟要高于可用区之间的网络延迟）的角度来说，都非常重要。此外，这种架构可以让应用程序支持各国不同的法规，例如欧盟安全港法规。[注1] 160

可用区不是数据中心

在一个 AWS 账户中，你基本可以认为两个不同可用区（例如 us-ease-1a 和 us-ease-1b）中的 EC2 实例，分别位于两个不同的数据中心中。

但是，当你使用多个 AWS 账户时就不一定了。如果你使用账户 1 在可用区 us-ease-1a 中创建了一个 EC2 实例，同时使用账户 2 在可用区 us-ease-1b 中创建了另一个 EC2 实例，实际上，这两个实例可能就位于同一个数据中心的同一台物理服务器上。

注 1　欧盟安全港是欧盟一组关于隐私的法规，用来管理欧盟公民对欧盟以外国家的数据传输行为。它通常关心的是如何存储数据才能符合当地的法律，而 AWS 区域的概念正是为此提供了支持。

为什么会这样？这是因为可用区名并不是静态映射到指定数据中心的，相反，两个不同的账户中，相同的可用区名（例如 us-ease-1a）可能代表了两个不同的数据中心。

当你创建一个 AWS 账户时，它会随机地创建一个可用区到指定数据中心的映射，[注2] 这意味着虽然都是“us-ease-1a”可用区，但是它们的物理位置可能相隔甚远。表 21-3 说明了这一点。这里我们展示了一个区域中的多个数据中心（从 1 到 8），然后演示了 4 个示例账户中，可用区与数据中心可能的映射关系。

表21-3：不确定的可用区映射关系

数据中心	AWS 账户 1	AWS 账户 2	AWS 账户 3	AWS 账户 4	…
DC #1	us-east-1a	us-east-1d		us-east-1e	…
DC #2	us-east-1a	us-east-1c	us-east-1a	us-east-1a	…
DC #3	us-east-1b	us-east-1a	us-east-1d	us-east-1d	…
DC #4	us-east-1c		us-east-1a	us-east-1b	…
DC #5	us-east-1d	us-east-1b	us-east-1c	us-east-1c	…
DC #6	us-east-1e		us-east-1b		…
DC #7			us-east-1e	…	
DC #8		us-east-1e			…

161 从这张表中你应当能注意到一些事情，首先，一个账户的一个可用区，实际上可能位于多个数据中心中。这意味着，你在同一个账户、同一个可用区内创建的两个 EC2 实例，有可能位于同一台物理服务器上，或者可能位于完全不同的数据中心中。其次，在不同账户中创建的两个 EC2 实例，即使它们的可用区不同，也不能确定它们是否位于一个数据中心。

例如，在表 21-3 中，如果账户 1 在 us-ease-1b 中创建了一个实例，账户 3 在 us-ease-1d 中创建了另外一个实例，这两个实例有可能都位于数据中心 3 内。

你需要牢记这一点，原因很简单，即使你在两个不同账户的不同可用区中创建了两个 EC2 实例，并不代表它们是互相独立的，也不能因此作为高可用的依据。

正如在本书的第 1 章、第 2 章所讨论的，保持冗余组件的独立性，对系统的可用性和风险管理至关重要。但是，当我们使用多个 AWS 账户时，AWS 的可用区模型并没有强制要求这一点，只有在一个 AWS 账号下的可用区模型才能做到这一点。

为什么你要使用多个 AWS 账户？实际上这个很普遍，很多公司会为不同的部门或者群组创建多个 AWS 账户，而 AWS 这么做可能是出于计费、权限管理或者其他原因的考虑。

注 2　实际上并不是随机的，而是由算法实现的。实际上，这个映射直到账户真正开始使用某个可用区或区域时才完成。

想知道他们为什么这样做吗

你有没有想过，每当AWS宣布出故障时，都会说故障影响了指定区域中的“某些可用区”，但从来不说是哪些可用区，为什么？

原因就在于系统的映射关系：如果我们假设4号数据中心有问题，对你来说可能是“us-ease-1a”，而对其他人来说可能就是“us-ease-1c”。他们无法给出具体的可用区名，是因为每个账户下的可用区名都是不一样的。

为什么AWS使用这种怪异的映射方法？一个主要原因就在于负载均衡。当用户要启动多个EC2实例的时候，如果这些实例都平均分布在多个可用区中，那么排在后面的可用区可能就没人去关心了。实际上，相比起“us-east-1e”，人们更愿意使用“us-ease-1a”作为常用的可用区名。这主要还是由人所决定的。如果AWS不采用这种人工映射的方式，按照字母表顺序，字母靠前的可用区就会使用过多，而字母靠后的可用区就会使用较少。通过这种人工映射的方式，AWS可以更有效地来平衡系统压力。

如何通过地理多样性真正做到高可用

162

你如何能确保使用的AWS资源一定有备份，并且该备份一定位于其他的数据中心，从而真正降低故障所带来的风险呢？

有一些方法可以做到。首先，确保你在同一个账户的不同可用区中维护了备份。如果备份在不同的账户中，确保你在每个账户内的多个可用区中都保留了备份。不要跨账户来比较可用区。

163

第 22 章

托管的基础设施

当提到云服务时你会想到什么？像大多数人一样，你可能会想到以下几点：

- 文件存储（例如 Amazon S3 服务）
- 服务器（如 Amazon 的弹性云服务，或者称为 EC2）

事实上，你最常用到的也就是这两种资源。

然而，云服务公司提供了其他各种托管服务来帮助你减轻管理上的压力，提高可用性，甚至提升可伸缩性。

了解如何组织、管理这些云服务，有助于你决定在应用程序中使用哪些功能。

基于云的服务架构

基于云的服务有以下三种基本类型：

- 原生资源
- 托管资源（基于服务器）
- 托管资源（不基于服务器）

图 22-1 展示了这三种类型。

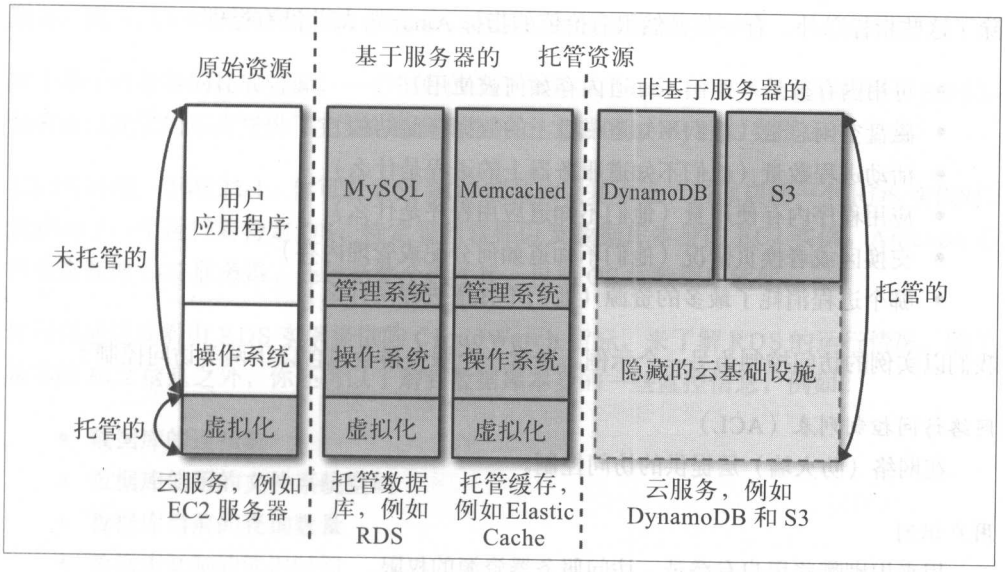


图22-1：基于云的服务类型

原生资源

164

原生的云资源为用户提供了一些基本能力，以及基础的管理功能。

原生云资源的一个例子就是 Amazon EC2，它以一种托管的方式提供了原生服务器的能力。

云服务提供了服务器虚拟化的基本管理功能，以及创建实例和初始文件系统的能力。但是，在实例启动并运行后，云服务厂商就无法了解到实例内部的操作了。

云服务厂商管理着实例输入 / 输出的数据流，以及 CPU 和 CPU 使用率。但是它们并不知道服务器上运行着什么，也不会监控任何内部的运行情况。

像 Amazon 这样的公司这么做是有目的的。服务器上运行什么是用户的事情，Amazon 并不想对这一点负责。Amazon 只负责到虚拟机这个范围，虚拟机内部的事情一概不管。

你可以从不同角度来观察这样做的结果。对于 EC2 实例来说，Amazon 会收集各种指标，并通过 CloudWatch 展现给你，其中包括：

- 流入实例的网络流量
- 流出实例的网络流量

165

- 磁盘数据的读取量
- 磁盘数据的写入量
- CPU 的使用率

除了这些指标以外，有一些显然很有价值的指标 Amazon 却并没有提供：

- 可用内存数量（他们不知道内存如何被使用）
- 磁盘空闲总量（他们不知道磁盘上的数据存储结构）
- 活动进程数量（他们不知道服务器上的进程是什么）
- 应用程序内存使用量（他们不知道应用程序是什么）
- 交换区或者换页情况（他们不知道如何分配或管理内存）
- 哪个进程消耗了最多的资源（他们不知道进程是什么）

我们以实例的访问控制为另一个示例。通常一个服务器会包含以下两类访问控制：

网络访问控制列表（ACL）
: 在网络（防火墙）层提供的访问控制。

用户识别
: 用来识别哪些用户有登录、访问服务器资源的权限。

Amazon 在创建实例后会负责管理网络 ACL（他们称其为安全组）。你可以在任意时间来修改安全组 ACL。如果你设置禁用某个端口，那么这个端口会立刻被禁用掉。如果你允许开放某个端口或者 IP 地址，这个端口或者 IP 地址也会立即生效。这些都是在网络层面完成的，在流量到达服务器之前进行流入 ACL 控制，当流量离开服务器后进行流出 ACL 控制。这个过程不需要访问服务器上的任何软件。

但是用户识别比较困难。当第一次安装服务器时，你可以指定一个会被安装在实例上的密钥对（公钥 / 私钥）。用户借助一些软件（例如，SSH[注1]），通过这个密钥对就可以登录到服务器上。但是，一旦实例被启动，Amazon 就无法再去添加和删除任何用户信息。
166 你无法从访问列表中增加或者移除密钥对，所有这些都由服务器上的程序（未托管）来完全控制，在 Linux 内核上就是指操作系统的 SSH 配置文件。

那软件如何升级呢？如果你的服务器上的软件需要升级的话，需要你自己去完成，Amazon 没有办法帮你做这些事情。

Amazon 可以帮助你管理服务器以外的所有事情，但是不负责任何服务器内部的管理。

托管资源（基于服务器）

基于服务器的托管资源，为指定的云服务提供了一个全栈的托管方案。

例如，托管的云数据库服务可以在已有的服务器上运行数据库和相关管理软件，使得云

注 1　Linux 操作系统上是 SSH，但是具体软件取决于镜像上安装的操作系统和软件。

服务厂商可以来管理整套方案、服务器以及运行在服务器上的软件。

对于基于服务器的托管资源，一个很好的例子就是 Amazon 的关系数据库服务（RDS）。该服务以托管的形式提供了完整的数据库解决方案，例如 MySQL 数据库。

我们再回顾一下图 22-1，你可以看到 RDS 的架构。基本上，当你启动一个 RDS 实例时，就启动了一个运行着特定 OS、特定管理软件以及数据库本身的 EC2 实例。Amazon 不但负责管理 EC2 服务器，还包括整个软件栈，包括 OS 和数据库软件。

你可以通过查看由 RDS 实例提供的 CloudWatch 指标，来了解 RDS 的运行情况。除了基本的 EC2 信息之外，你还可以了解到数据库本身的一些监控信息，例如：

- 数据库的连接数
- 数据库使用的文件系统空间
- 数据库当前的查询数量
- 数据库复制的延迟时间

这些都是只能从 OS 或者数据库软件获取到的指标。

另外一个了解服务的方式就是学习你可以操作的各项配置。除了可以配置服务器的基本信息（例如网络连接数和挂载磁盘）以外，还可以配置一些数据库本身的信息，例如最大连接数、缓存信息，以及其他配置和性能调优参数。

除此之外，软件升级也是云服务厂商负责的一部分工作。如果某个数据库软件需要升级，Amazon 会替你完成升级操作。 167

托管资源（不基于服务器）

不基于服务器的托管资源提供了某些特定功能，但是不会暴露提供该功能的底层基础设施情况。

对于不基于服务器的托管资源，一个很好的例子就是 Amazon S3 服务。这个服务提供了基于云的文件存储和传输服务。当你在 S3 上保存一个文件时，你会直接和 S3 服务进行通信，Amazon 不会为你分配任何服务器。实际上，这时可能会有一台或多台服务器[注2]在背后为你完成操作。

整个执行过程是可以托管的，但是你只能查看文件，以及调用暴露出来的服务接口（例如在 S3 上进行上传、下载、删除文件等操作）。你无法了解或控制任何服务底层运行的操作系统或者服务器。这些服务器由所有使用服务的用户共享使用，由 Amazon 统一进

注 2　如果是 S3 服务，实际会有很多很多的服务器。

行管理和控制，无须用户干预。

使用托管资源的影响

使用某个托管服务或者其中一部分功能，会给你带来很多好处，主要包括以下几点：

- 你不需要安装或者升级托管系统上的软件。
- 你不需要调整或者优化系统（但是你可以通过云服务厂商去进行调优）。
- 你不需要监控或者保证软件运行的性能。
- 你不需要安装任何软件，云服务厂商就可以提供一些所需的监控数据。
- 云服务厂商可以为服务提供备份和复制的功能。

168 但是托管服务也有不足之处：

- 不能改变软件的运行机制。
- 无法控制何时、如何去升级软件，以及当前运行的软件版本。[注3]
- 只能使用由云服务厂商提供的监控数据和配置项。

使用非托管资源的影响

如果你不使用托管服务，也有一些好处，主要包括以下几点：

- 你可以控制运行什么软件，运行什么软件版本，以及如何进行安装。
- 你可以控制软件升级的时间和方式，以及决定是否需要升级。
- 你可以按照需要的内容和方式来监控和控制软件。

当然，使用非托管资源也有以下弊端：

- 没有免费的午餐，你需要自己负责系统的所有管理和维护工作。
- 你需要自己实现数据备份和复制机制。
- 你需要自己监控软件，以保证它们正常运行，否则无人会知道故障发生。
- 如果软件运行出现问题，也只有你自己能够去修复它。云服务厂商无法提供任何帮助。

监控和 CloudWatch

我经常听到一些疑问："CloudWatch 是否完全能够用来监控云服务资源的使用情况"，我

注3 不过，云服务厂商也会提供某些功能。例如，RDS 提供了支持的数据库版本列表，但并不支持所有版本。但是对于像 S3 这样的服务，你根本无法控制软件的升级过程。

的回答是“不能”。

为什么？因为 CloudWatch 只负责 Amazon 自己服务的监控指标，因此它可以提供对 DynamoDB、RDS 和 ElastiCache 等服务的很好监控。 <169>

但是对于 EC2 服务器，CloudWatch 只能提供一些基本的服务器和虚拟化监控指标。对服务器内部使用情况它提供不了任何监控或者报警。

如果需要全面监控你的 EC2 服务器，还需要以下两种工具：

- 监控服务器内部操作系统的工具，用来提供例如内存使用率、交换区和文件就使用率等信息。
- 内部监控系统运行状况的工具，用来提供应用程序运行情况、用户使用情况，以及系统内部的错误和问题等信息。

只有以上三种监控功能结合在一起（以上两点和 CloudWatch），才能对一个 EC2 服务器进行完整监控。

171

第 23 章

云资源分配

为了有效并且高效地使用云服务资源，你需要理解它们是如何分配、使用和计费的。云资源通常可以被划分为以下两类：

- 固定额度的资源
- 按需分配的资源

固定额度的资源分配

固定额度的资源指的是已经预分配在分散单元中的云资源。你提出需要多少某种资源，然后系统帮你分配。资源总量是按照你的要求进行预分配，而不是按照你的实际使用情况来实时分配的。

固定额度的资源有以下一些特点：

- 分散在不同的单元中。
- 你需要指定需要多少资源，系统为你进行分配。
- 如果系统使用的资源少于所分配的资源，那么就会剩下一部分未使用的资源。
- 如果系统需要更多资源，那么预分配的资源可能会不够用。
- 需要合理计划资源的使用情况，避免过度分配或者分配不足的情况出现。

固定额度资源的典型例子就是服务器，例如 Amazon EC2 实例。你可以指定需要的实例
172 数量以及服务器大小，由云服务将它们分配给你使用。此外，一些托管的基础服务，例如云数据库[注1]也使用这种分配方式。对于这些服务，你只需指定服务的单元数量以及单元大小，云服务会帮你分配好资源。

注 1　例如 Amazon RDS、Aurora 和 ElastiCache。

不过也有一些固定额度资源，在操作上会略有一些不同，例如Amazon的DynamoDB服务。在使用DynamoDB时，你需要指定DynamoDB表的容量，这样就不是按照服务器的大小来计算，而是按照吞吐量来计算容量。你指定要为表分配多少额度，它们就可以使用多少。如果你没有使用这么多额度，剩下的就浪费了。如果系统的使用超出了分配的额度，就会出现资源不足，直到被分配更多的额度。因此，虽然看上去不同，但其实它们也是按照与服务器相似的方式进行分配的。

调整分配

通常情况下，额度都是按照一定梯度进行分配的（例如服务器按照每小时计算，DynamoDB的容量也按照每小时计算）。你可以调整分配给应用程序的服务器数量，或者分配给DynamoDB表的容量，但是只能按照一定的梯度调整（例如服务器的大小，或者容量单元的大小）。虽然有不同的梯度可以选择（例如大小不同的服务器），但你每次必须分配整个服务器或者整数容量。

你有责任确保手头拥有充足的资源。这类似于传统的数据中心对服务器的容量规划。你需要先按照当前需求去分配容量，等到确认容量需求有变化后再重新分配。这是传统的、不基于云分配服务器的方式。

但是，云上分配比传统数据中心的容量分配更加简单。因此，我们可以使用一些算法来提升分配策略。比如，因为分配变更可以（几乎）立即生效，所以你可以等到目前的资源几乎用完时再去增加容量。

另一种办法就是定期去调整分配额度。比如，可以在白天业务负载较高的时候增加服务器数量，而在晚上负载较低的时候减少服务器数量。

还有另一种方法，就是根据当前资源的使用情况，动态、自动地调整分配容量。比如你可以监控服务器上CPU的使用率，当上升到一定阈值时就自动增加服务器。[注2] 你可以将这些自动化机制整合到应用程序或者基础服务中，或者利用云服务（例如AWS的自动扩容服务）根据指定条件的当前情况自动进行调整分配。

173

无论选择哪种方式去调整资源分配，要记住一点，已分配的容量就是你的可用容量，有可能部分容量被浪费掉，但更糟的是发生资源不足的情况。即使你使用自动扩容服务，也并不意味着它能够及时发现资源不足的情况，例如突发的大量资源占用。

注2　或者反过来，当下降到一定阈值时就减少服务器。

分配问题

以 Amazon 的弹性负载均衡（ELB）为例，它会根据当前的流量自动扩展，为应用提供负载均衡的功能。如果请求的流量很小，ELB 就会减少用于负载均衡的服务器数量，相反，如果请求流量变大，ELB 会自动增加更多的服务器来处理请求。这些对用户来说都是自动并且透明的。这就是 ELB 如何能够收取较低的入门费用，但是又可以在流量激增时自动扩容（增加相应的价钱）。这样你可以在流量低峰时节省费用，在流量高峰时自动扩容。

但是，在某些场景下，这种自动分配机制可能会带来一些副作用。如果你的应用因为一些社交媒体活动突然走红，流量突然达到一个高峰，负载均衡器可能来不及迅速进行扩容。那结果会怎样呢？在请求量升高一段时间之后，负载均衡器的资源可能逐渐不足，导致页面请求响应变慢，或者无法响应，从而降低用户体验。虽然当 ELB 发现流量增加之后会自动增加处理负载均衡的机器，自动改善资源不足的情况，但是这种扩容通常需要几分钟才能完成。在这个过程中，用户的体验会下降，系统可用性也会受到影响。

174

为了避免这种影响，你可以联系 Amazon 的客服代表，提前预警可能到来的流量变化，这样他们可以提前对负载均衡器进行预热。[注3] 这样可以在流量高峰到来之前，提前对负载均衡器进行扩展（使用更好和更多的服务器）。不过这只能对一些预料流量增加的情况有效，对毫无征兆的突发情况不起作用。

这种情况是这类云资源分配的问题之一。

无论你使用手动还是自动的方式调整资源分配，依然会受到预分配额度的限制。不管你是否使用，都要为所有分配的资源付费。如果应用对资源的需求超过了当前额度，就会产生资源不足的情况。因此，不管是手动还是自动的方式，合理的用量规划对管理资源都非常关键。

预留容量

通常你可以随时来调整容量分配，[注4] 根据需求去增加或者减少资源。

这就是云服务的优势之一，如果当前每小时需要 500 台服务器，而下一个小时只需要 200 台，那么你只需要支付一小时 500 台和一小时 200 台的费用。这种方式简单明了。

注 3　更多信息请参考文章《ELB 最佳实践》（*http://amzn.to/28Jgwjz*）。
注 4　有些服务有限制，例如 DynamoDB，对调整容量的频率有一定限制。

但是，由于分配资源容量所赋予的灵活性，你会为此多花一些钱。

如果你的需求很稳定怎么办？如果你经常需要至少 200 台服务器怎么办？当你对服务器需求很稳定的时候，为什么要为按小时计费的灵活性付费呢？

于是预留容量出现了。预留容量是指，你预先向云服务厂商承诺在一段时间内（比如 1~3 年）使用一定量的资源。作为交换，你可以享受到一定的优惠。

示例23-1：预留容量示例

预留容量并不会限制分配资源的灵活性，它只是向云服务厂商保证能够使用一定量的资源。

假设你的应用程序平时需要 200 台服务器来支撑，不过在流量高峰时需要扩容到 500 台。你可以使用自动扩容功能去动态调整服务器的数量，因此服务器的使用数量会在 200 到 500 之间动态变化。

因为你一直都需要使用至少 200 台服务器，所以你可以购买 200 台服务器的预留容量。比如你购买了 1 年 200 台服务器的使用量，这 200 台服务器的价钱相对较低，虽然你需要一直付这么多钱，但是还好，因为你一直都需要使用这么多服务器。

至于另外的 300 台服务器（500-200），你可以按以小时计费（价格更高）的方式付费，不过只是为使用的那些时间付费。

通过交换对预留资源的承诺，你可以较低价格获得这些资源。[注5]

基于使用量的资源分配

基于使用量的资源不是被分配的，而是按照应用程序的使用率来计算的。它们无须分配，只会按照实际使用量计费。

基于使用量的云资源有以下特征：

- 没有资源分配的步骤，也无须进行容量规划。
- 如果应用需要更少的资源，你就可以使用更少的资源，收费也更低。
- 如果应用需要更多的资源，你就可以使用更多的资源，收费也更高。
- 在合理范围内，你不需要做任何事就可以将使用量从一个极小值扩大到一个非常大的值。

注 5　使用预留资源还可以保证这些资源分配在你想要的可用区中。如果不使用预留资源，当你希望在指定可用区中创建某个实例时，AWS 可能无法满足这一点。

176 • “合理范围”由云服务厂商根据其能力来统一定义。

通常情况下你不需要知道这些资源是如何分配、扩容的，这些对你都是不可见的。

Amazon S3 就是一个经典的基于使用量的云资源。S3 服务是按照你存储和传输的数据量来计费的。你根本不用去预先估计需要多大的存储空间，或者多大的数据传输量。无论你需要多大的量（在系统限制范围内），S3 都能支持，你只需要为实际的使用量付费。

基于使用量分配资源的好处

因为不需要提前规划容量，所以这种服务易于管理和扩展。这种基于使用量的资源才是云服务带来的真正好处之一。这也得益于云服务多租户的特点。

在像 Amazon S3 这样的服务背后，是海量的磁盘存储系统和海量的服务器，按照每个用户的每个请求按需分配。如果你的应用程序对资源有需求高峰，就可以从一个共享的可用资源池中分配。

这些可用资源池被所有的用户共享，所以底层这个资源池可能非常巨大。当你的资源使用高峰逐渐消退时，另外一个用户的使用高峰可能刚刚开始，这样本来服务于你的资源就会被分配给另一个应用程序使用。这些资源转移再分配的过程对你来说都是透明的。

只要可用资源池足够大，就足以处理所有请求，以及满足所有用户的使用高峰，不会出现资源不足的情形。服务的规模越大（使用服务的用户越多），对云服务厂商来说，平衡资源使用高峰和计划足够容量的能力也就越强。

只要没有单个用户占用大量可用资源的情况出现，这种模型就是有效的。如果某个大户占用了资源池中的大部分资源，那么他在高峰期间就会遇到资源不足的情况，并且也会影响其他用户可以使用的容量。

177 但是像 Amazon S3 这样的服务，它们的规模如此巨大，[注6] 以至于没有用户可以用掉大部分资源，所以它的资源分配机制依然成谜。

注 6 根据 Amazon 最新发布的相关数据，2013 年 S3 系统存储量已达 2 万亿个存储对象，如果把这些存储网络比喻成银河系，每个存储单元是一颗星的话，那么每颗星可以容纳 5 个存储对象。具体请参考该资料：Amazon S3 对象存储量已达 2 万亿，110 万次请求 / 秒（*http://amzn.to/28Jw1HJ*）。

然而，即使是 Amazon S3 也有自己的限制。如果你的应用系统需要存储或者传输极其海量的数据，那么就会遇到一些 S3 的限制，目的是防止造成其他用户的使用资源不足。因此，大量使用 S3 资源的服务会遇到这些人为限制，并且自己也会遇到资源不足的情况。这种情况一般只会发生在 PB 级数据的存储和传输场景下。

即使你真的需要使用如此多的 S3 资源，也有一些办法能够突破这种限制。除此之外，你还可以联系 Amazon 申请提高限制的阈值。他们会根据你的需求，将这个值上升到合理的范围，并将其加入到后续的容量规划中，以此来保障有足够可用的容量来满足你或者其他用户的需求。

资源分配技术的利与弊

如表 23-1 所示，我们之前讨论的每种技术都有相应的利弊。

表23-1：云资源分配技术之间的比较

	固定额度分配	基于使用量分配
服务示例（Amazon AWS）	EC2，ELB，DynamoDB	S3，Lambda，SES，SQS，SNS
需要额度计划	是	否
收费依据	分配的额度	花费的额度
未使用的情况	额度闲置	N/A
超出使用的情况	应用程序资源匮乏	N/A
是否可以预约额度以节省开支	是	否
如何增加额度	手动或者通过脚本来变更额度，可以延迟变更	自动并及时变更
如何处理用量峰值	允许可能的用量匮乏，或者提高额度	无感知处理
如何处理超出的额度	已分配，并且只能供你使用	放到全局资源池中，可供其他客户使用

178

179

第 24 章

可伸缩的计算选项

安装、配置、管理服务器，然后将应用程序部署在上面，这只是部署应用程序的一种方式。除此之外，还有很多其他的方式，各有利弊。在这一章中，我们会讨论其他的部署方式。我们会从基于云服务的服务器部署，一直讨论到的 Amazon AWS Lambda，看看它们之间的区别和相似点，以及你如何选择适合的方式。

在这一章中，我们会比较各种基于云服务的应用程序运行环境，[注1] 其中包括以下几项。

云服务器

这属于比较基础的服务器技术，例如 Amazon EC2 服务器。

计算分片

这是在与服务器独立的计算环境中运行传统应用程序的方式，例如 Heroku Dynos 和 Google APP Engine。

动态容器

包装了完整的服务器功能，提供了快速启动、停止服务以及迁移服务到新系统的能力。Docker 是典型的例子。

微计算

体积小、高度可扩展、基于事件驱动的运行环境，AWS Lambda 是最好的例子。

180 在比较各种方式之前，我们先了解一下它们各自的特点。

注 1　由于其中一些技术也适用于非云的部署环境，但是我们讨论的主要是伸缩性，所以只关注基于云服务的使用场景。

云服务器

云服务器是实现可伸缩计算最简单的途径，也是最贴近于传统的软件开发和系统架构模型的方式。云服务器可以应用在很多地方，Amazon 的 EC2 实例就是最流行、最广为人知的例子。

使用云服务器最大的优点，就是它们可以快速上线和使用，对于应用扩展非常友好。但是，这对应用程序的架构有一定要求，真的能做到增加服务器可以实际提高应用程序的可伸缩性。虽然很多应用都能做到这一点，但不是所有应用都能做到。

优点

- 单计算周期成本最低。
- 功能限制极少，可以使用服务器的绝大多数能力。
- 可持续运行。

缺点

- 固定额度分配。[注2] 所谓的扩展，就是发展新的服务器，并在其上部署应用程序。
- 你需要为分配的容量付费，而不是实际使用的容量。
- 应用程序需要在架构上支持服务器的水平扩展。
- 在集群环境中，无法有效利用单台服务器的很多优点（例如本地存储资源）。
- 应用负责人必须管理和维护服务器，包括软件和安全策略升级。

适用场景

云服务器可以用于各种常规应用，也能够满足大多数的伸缩要求。

计算分片

181

计算分片是另一种执行模型，用户无须关心他们运行的服务器。在该种模式下，用户需要将应用程序部署到一个 PaaS（平台即服务）平台上，以托管的方式运行。

在众多基于计算分片的计算引擎中，Heroku Dynos 是一个典型的例子。

优点

- 易于在相对细粒度的规模上分配容量。

注 2　请参考第 23 章对资源分配优缺点的讨论。

- 这种云计算厂商的策略都是过度分片，这可能会帮助降低单个分片的成本，对于低流量的应用程序尤其有用。
- 不需要管理服务器。

缺点

- 单位计算周期要比基于云服务器的方式更高。
- 固定额度分配。所谓可扩展就是配置新的分片来运行应用程序。
- 你需要为分配的容量付费，而不是实际使用的容量。
- 你无法控制当前使用的服务器和基础设施。

适用场景

一个典型例子就是低流量的 Web 应用。

这种方式最适合于运行传统应用程序，但是又不想管理服务器或者其他基础设施。

如果应用的负载比较低，那么只需要几个分片就可以运行整个应用，从而起到节约成本的作用。但是，相比基础的服务器而言，应用程序越大，这种方式的成本越高。

值得注意的是，谷歌的 APP Engine 是另一个基于计算分片的解决方案，除了付费机制不同，以及资源分配更加灵活之外，其他的优缺点都与我们所说类似。重点是，许多计算分片的优缺点都不是来自于概念，而与具体的实现有关。

182

动态容器

动态容器是容器技术的特定应用，它可以动态分配资源，在不同服务器之间迁移容器，从而提供更高的可伸缩性和托管计算环境。注意，这些计算环境中都可以使用容器作为部署方式。这里我们指的是一种易于迁移和调度容器的动态模型，能够优化系统资源，使得添加新容器变得更加容易。

在动态容器环境中，应用程序被整体包裹在容器中，可以快速简单地实现启动、停止和重定位。通过与微服务架构的应用程序相结合，动态容器可以提供一个高度动态和灵活的应用程序，简单有效地进行伸缩，而且相比传统计算或者静态容器部署等方式，它可以更加有效地优化现有计算资源。

优点

- 价格便宜，并且能够有效优化当前服务器的功能。

- 更容易扩展，因为容器可以动态部署在任何有计算资源的地方。
- 容器对应用程序的限制很少，可能较慢的应用启动时间会降低容器调度的优势。
- 容器可以一直运行，或者按需运行，节省资源。
- 容器很容易部署，服务器只需要极少的配置就可以启动容器。
- 启动一个服务变得很容易，而且可以做到自动化。

缺点

- 你需要去主动管理容器，并且需要一些软件对容器进行部署和调度。一些云服务厂商提供了这类相关服务，但是这仍然是一块较新的领域，缺少领先的动态容器管理解决方案。容器调度和管理能带来的优点还未被人们充分认识到。 183

适用场景

动态容器能够简化基于微服务架构的应用程序的部署，还能够快速启动和关闭服务实例（容器的启动和停止，与 Linux 进程的启动和停止非常相似）。

微计算

微计算是可伸缩计算技术的巅峰。微计算模型可以执行一段相对简单的（微）代码，并且可以用于响应某些事件，例如用户请求或者服务调用。每个事件都会触发并执行一个与事件有关的微计算函数。在任一时刻，你可以独立执行任意数量的微计算函数，处理所有出现的请求。这有效提供了一种几乎可无限扩展的运行时环境。AWS Lambda 就是微计算技术的典型例子。

微计算和计算分片在程序执行的方式上类似，二者最大的不同在于，计算分片是固定额度分配，且在不同单元上调度。虽然你可以修改当前单元的数量来处理请求，但是最终能够处理的请求数，受限于计算分片的扩展能力。微计算是完全动态扩展的，只需在多个实例上运行处理当前请求的函数即可。

为了达到以上目的，Lambda 中的计算函数必须简单灵活，而且不需要初始化或者销毁，以便运行起来更有效率。

优点

- 自动化且近乎无限的扩展能力。[注3]
- 无须管理服务器，无须系统升级。

注 3　显然并不是真的无限扩展，只是对于任意实际的需求，可以认为扩展能力接近无限。

- 根据事件处理的实际数量进行计费，成本可控。
- 可容易地支持快速的伸缩性变化（扩展或者收缩对用户都是透明的）。

184

缺点

- 功能和支持的语言非常有限。
- 应用程序必须围绕微计算技术去架构。
- 单个计算周期价格较高，但是定价模型和细粒度的计费模型可以缓解价格。
- 当前的一些实现方案（AWS Lambda）没有提供良好的部署、回滚，或者 A/B 测试功能，使得一些要求高可用的应用需要采用其他架构或者外部工具才能够使用。

适用场景

微计算针对海量数据流计算和事件处理进行了优化，非常适用于数据验证、转换和过滤方面。在网络边界中，它适合用于输入数据的校验工作。

它最适用于事件驱动的场景下，通过简单的代码片段对指定事件进行处理。

如何选择

现在有各种可扩展的计算方案供我们选择，几乎任何应用程序和 IT 运维需求都能找到一种或多种替代方案。

但是我们面对这些技术方案应该如何去选择？由于其中很多技术都是由特定的云服务厂商提供的，因此，对这些技术的选择同时也是对云服务厂商的选择。

尽管如此，几乎任何一家主流的云服务厂商都支持了多种方案，这也是当今云服务的优势之一。不用担心在一个应用中使用多种方案，因为应用的不同部分有不同的需求。你只需像往常一样，从应用程序的可用性和可伸缩性去考虑，然后选择与应用最匹配的方案，以及能支持该应用程序的团队。

第 25 章 AWS Lambda

AWS Lambda 是由 AWS 新推出的一个软件执行环境，主要提供事件驱动的计算能力，无须购买、安装、配置或者维护服务器。Lambda 为你提供了几乎无限的扩展能力，并且支持亚秒级的计费。

Lambda 非常适合于事件触发行为的后台处理，以下是一些常见的使用场景：

- 新上传图片的格式转换。
- 实时的指标数据处理。
- 流式数据的验证、过滤和转换。

它适合于以下任意一种形式的数据处理：

- 应用程序中某个事件发生后需要进行一些操作。
- 数据流需要进行过滤或者转换。
- 对传入数据进行必要的边界值校验。

不过，Lambda 真正强大的地方还是在于它彻底地解决了伸缩性的问题，它可以在不需要任何操作的情况下，扩展到几乎任意的规模。

使用 Lambda

此刻你可能会想到，“这听上去很棒，但是什么样的架构适合使用 Lambda？”接下来的几节内容我们来解答这个问题。

事件处理

以一个图片管理程序为例。用户可以将图片上传到云服务并存储在 S3 中，应用程序会

显示图片的缩略图，并且让用户更新图片的相关属性，例如名称、地点、图片中的人名等。

这种简单的应用程序就可以使用 AWS Lambda 对用户上传到 S3 后的图片进行处理。当用户上传一张图片后，会自动触发一个 Lambda 函数，为该图片生成一个缩略图并上传到 S3。同时，另一个 Lambda 函数会获取图片的多个特征（例如尺寸、分辨率等）并将这些元数据存储在一个数据库中。随后，图片管理程序可以提供操作数据库中元数据的功能。

这个架构如图 25-1 所示。

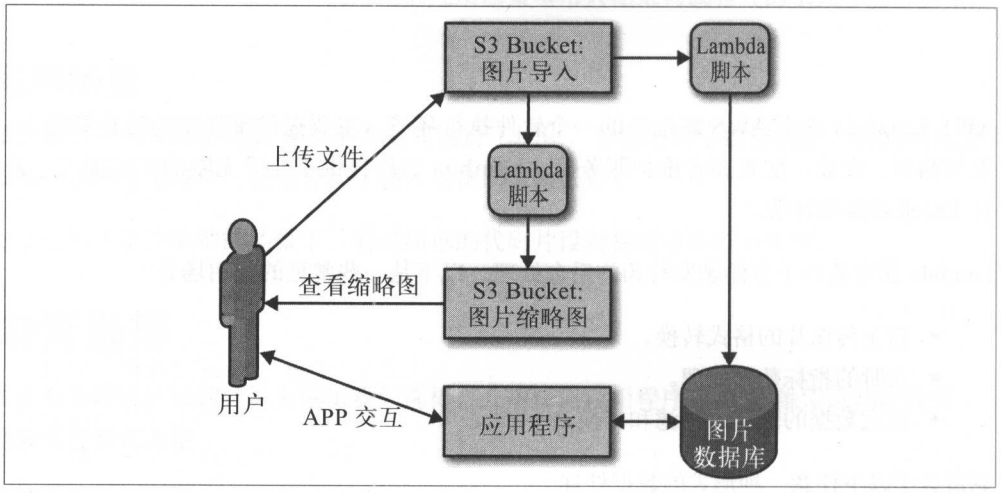


图25-1：文件上传Lambda的用法

图片管理程序无须参与图片的上传过程，它可以依赖于标准的 S3 上传功能和这两个 Lambda 函数，由它们完成文件的所有上传过程。这样，图片管理程序只需要考虑它所擅长的事情：操作已有图片在数据库中保存的元数据。

手机应用后台

假设有一个手机游戏，在云服务上存储了用户的游戏进度、奖品以及高分榜，这些数据用于共享社区以及个人用户的移动设备上。

187 这个应用程序需要在后台创建一系列 API，来完成将数据存储到云服务上、从云服务中获取用户信息，以及进行社区交互的功能。

这些 API 都是通过一个关联了许多 Lambda 函数的 API 网关来创建的。[注1] 这些脚本会执

注 1　Amazon 的 API 网关服务是一个与 AWS Lambda 配合创建 API 的服务。

行一些必要的数据库操作，来处理游戏的云服务后端工作。

该架构如图 25-2 所示。

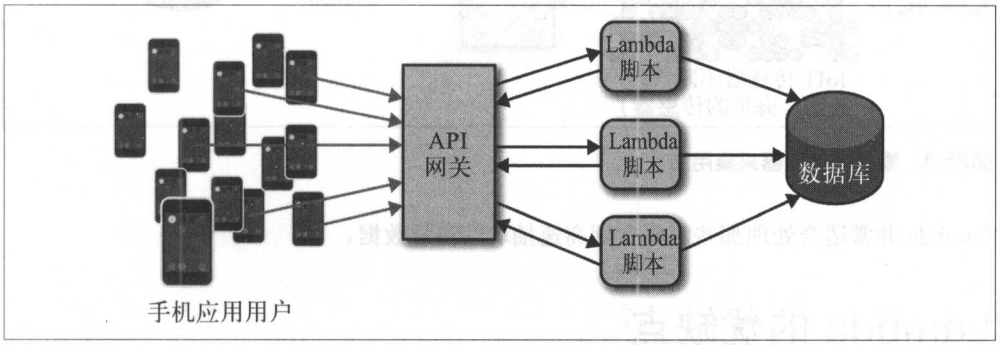


图25-2：手机后台Lambda用法

在这个例子中，后端不需要任何服务器，并且所有扩展都是自动处理的。

物联网数据采集

假设有一个应用程序，可以获取全世界各地大量的传感器数据。由于这些传感器会定期向应用程序发送数据，所以应用的服务器端会定期收到大量的数据，需要保存在某种数据存储器中。这些数据由某些后端程序所使用，不在这个例子里详细讨论它们。数据采集模块需要对数据进行验证，可能会执行一些简单的数据处理，然后将结果保存到存储器中。

这个程序只是简单执行一些基本的数据验证操作，然后将数据保存下来供以后使用。但是，虽然应用程序比较简单，但是必须以巨大的规模运行，才能够支持每分钟百万甚至十亿次的数据采集，而实际的规模取决于传感器的数量。

该架构使用了数据采集流水线[注2]，将数据发送给一个 AWS Lambda 函数，由该函数在数据存储前进行必要的过滤和处理。 188

该架构如图 25-3 所示。

注 2　Amazon Kinesis 是一个实时的流式数据采集流水线，用来处理大量的数据流请求。

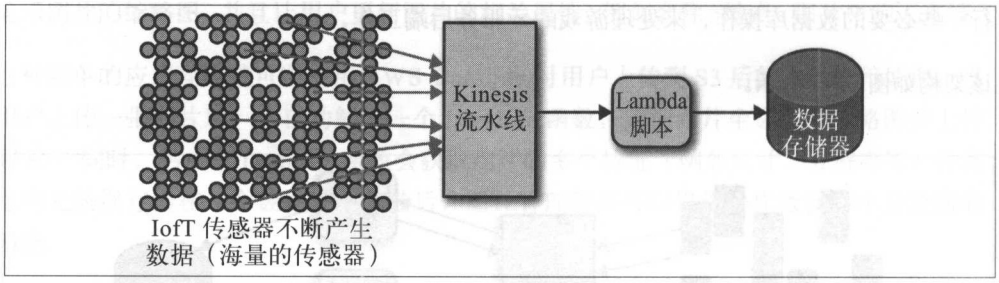


图25-3：物联网传感器采集用法

Lambda 非常适合处理那些需要定期高速抽取的海量数据。

Lambda 的优缺点

AWS Lambda 有一个主要的优点：可伸缩性。Lambda 非常善于处理大流量负载，同时无须扩展应用程序所在的底层基础设施。

为了达到这一点，它需要代码本身运行起来极其简单，这样才能方便地在多台服务器上快速、高效地并行执行，按需伸缩。

这就是 Lambda 的核心：以超大规模运行小型代码片段。

那么什么场景下你应当使用 Lambda 呢？为了回答这个问题，我们先了解一下 Lambda 的缺点：

- 隐含的编程需求（简单、事件驱动、快速处理）。
- 复杂的配置和安装。
- 没有原生的内置预发布或测试环境。
- 没有原生的部署 / 回滚功能。
- 没有原生的 A/B 测试功能。
- 没有能够开发、测试 Lamda 函数的开发环境。

189 简而言之，Lambda 在小型脚本的伸缩性上做得很好，但是不适合用来部署大规模的应用程序。

如果用得好，AWS Lambda 可以极大帮助你满足规模上的极度扩展。但是，你需要将它用在适合的场景下，对于那些超出 AWS Lambda 能力范围以外的需求，请使用其他的部署和执行方式。

第Ⅳ部分

总结

可伸缩的架构不只是用来处理大量的用户。

193

第 26 章

融会贯通

我们在这本书里涉猎了很多主题和内容，如果将它们结合到一起，就可以帮助你去扩展应用程序：

- 可用性和可用性管理
- 风险和风险管理
- 使用服务和微服务来构建应用程序
- 扩展你的应用程序及其开发团队
- 使用云服务来帮助你扩展应用程序

可用性

可用性是指应用程序完成其应有任务的能力。它和可靠性不同，可靠性是应用程序不出错的能力。一个系统如果计算 2 + 3 的结果是 6，说明可靠性很差，如果计算 2 + 3 却从来不返回结果，说明可用性很差。

可用性差通常会因为很多原因引起，例如：

- 资源枯竭
- 计划外的负载变化
- 流动行为的增加
- 外部依赖
- 技术债务

194 除了应用程序本身的功能之外，可用性通常是程序尝试去扩展的首要因素。关于可用性的话题在第 I 部分已经做了广泛的讨论，包括如何去衡量可用性以及如何去维护一个高可用程序。即便你的应用程序在持续不断地扩展，其可用性也可以根据需要维持在一个合适的级别。

风险管理

如果你不能识别系统中有哪些风险，就无从进行风险管理，这也是本书第 II 部分的重点内容。你必须清楚地理解，风险是运行一个高可用以及高可伸缩应用程序的首要条件。

如果你已经清楚风险是什么，就必须开始管理这些风险。虽然我们希望消除所有的风险，但是从实际成本和机会成本的角度来看，这样做的代价非常高。你肯定有很多更重要的、用户更关注的问题需要去解决，它们对于用户、公司以及你自己所带来的利益，远比消除应用程序中每个风险大很多。

相反，你应当管理风险而非消除所有风险，这涉及评估每个风险的两个值，即风险发生的可能性和严重性。这两个值非常重要，所以我们专门在第 6 章详细进行了讲解。一般来说，严重性指的是风险发生后造成的损失，而可能性指的是风险发生的几率。

如果一个风险可能会对应用程序带来很严重的后果，但是实际上很少发生，那么你可能没必要去消除它。同样，如果一个风险极有可能发生，但是对你的程序影响很小，那么它也不是需要优先消除的。实际上，最需要你解决的风险，是那些既有可能会发生，并且可能造成相当严重后果的风险。

因此，我们引入了一个称为“风险模型”的工具，它可以有效地帮助你管理应用程序中的风险，并且确定哪些风险需要缓和或者消除。

我们讨论了一些用来缓和风险的技术，以及一些如何验证缓和执行计划和构建低风险应用程序的技术。

服务

服务是一个有明确边界的系统，为构建大型产品提供业务功能。服务提供了一种应用架构的模式，可以在构建系统时提高系统和开发团队的可伸缩性。

当构建大规模应用程序时，服务可以降低规模预估的难度，帮助团队调整关注点和控制力，降低本地开发的复杂性，以及提高测试和部署的能力。 195

为了解决如何在服务层面构建高可用性的问题，以及降低服务失败对应用程序和用户的影响，我们为你提供了一些工具和建议。

扩展

我们提到了你应当如何制订扩展计划，来保证系统的可用性和管理能力，同时讨论了构

建系统时要注意“两次失误的高度”，以避免连续性的故障和级联依赖的故障。

我们还介绍了“由独立团队负责的服务架构”模型，即 STOSA。当你的应用程序规模需要扩展的时候，它提供了一个能够扩展开发团队组织规模的模型，可以让大量的工程师在一个应用程序上高效地协同工作，但同时又不会降低该程序的伸缩性和可用性。这需要你围绕着这些原则，来定义服务所有者的职责以及管理应用程序。

我们也讨论了如何使用内部 SLA 和服务分级等工具来管理服务依赖，这样即使在高速增长的时期，也能够保证应用程序的质量。

云服务

最后，我们介绍了云服务，以及如何使用云服务来构建高可伸缩的应用程序。

我们介绍了云服务如何改变了我们对计算的理解，以及构建应用程序的思考方式。随后，我们讨论了如何使用云服务为应用程序增加地理上和网络上的拓扑多样性，以及避免在这个过程中由于内部依赖所带来的一些陷阱。

我们着重介绍了托管基础设施，以及如何通过它来构建高可伸缩的应用程序。还介绍了如何分配基于云的资源，以及如何确保应用程序有足够运行的云资源。

随后，我们讨论了使用云服务时可选择的计算方案，也介绍了 AWS Lambda，以及它为可伸缩性方面带来的革命性未来。

196

面向可伸缩的架构

设计一个应用程序的可伸缩性架构，并非仅仅为了能够处理更多的用户，它还包括很多其他需要扩展的方面：

- 必须能够处理大量且不断增长的用户。
- 必须能够处理大量且不断增长的用户数据。
- 必须能够应对日益复杂的用户需求。
- 必须能在公司扩张时，让更多的开发人员一起来开发应用程序，而且必须是在不牺牲开发速度、效率或者应用程序质量的前提下。
- 即使前面列举的所有要求变得更高，你也必须保证应用程序在线上正常运行。

这些问题并不容易解决，但是本书中所讨论的很多技巧就是为了帮助你解决它们以及其他伸缩性方面的问题。

索引†

†： 索引所列页码为本书英文版页码，请参照正文侧边用“⟨⟩”表示的原书页码。

S

T

U

V

关于作者

Lee Atchison 是 New Relic 公司的首席云架构师和布道师。他已经在 New Relic 工作了 4 年，负责设计并领导建立了 New Relic 的基础设施产品，帮助 New Relic 搭建了健壮的服务化系统架构，支撑起公司从一个很小的 SaaS 创业公司成长为一个高流量的公众企业。他非常擅长构建高可用的系统。

Lee 拥有 28 年的行业工作背景，之前在 Amazon.com 担任了 7 年高级经理，了解到如何搭建基于云的、可伸缩的系统架构。在 Amazon，他领导并建立了公司第一个软件下载商店，搭建了 AWS Elastic Beanstalk 服务，并带领团队将 Amazon 的零售平台从一个单体架构成功迁移到了基于服务的架构。

封面介绍

本书封面的动物是一只纺织锥形蜗牛（也称织锦芋螺）。由于其贝壳上独特的黄棕色和白色特征，所以它又被称为“金锥布”，它通常能够长到 3 到 4 英寸。纺织锥蜗牛生活在红海的浅水中、澳大利亚和西非的海岸边，以及印度和太平洋的热带区域。

同其他芋螺类动物一样，纺织锥蜗牛也是食肉动物，捕食其他蜗牛，通过从“齿舌”（一个类似于小针一样的口器）注入毒素将猎物杀死。从其身上提取的“芋螺毒素”毒性非常强，能够导致麻痹或者死亡。

纺织锥蜗牛一次可产下几百枚蛋，自由成长为成年蜗牛。虽然它们的壳有时被作为装饰物贩卖，但是它们的数量巨大，不属于濒危物种。

O’Reilly 书籍封面上的许多动物都是濒危动物，它们对于世界来说非常重要。如果你想了解能为它们做什么，请访问 *animals.oreilly.com*。

本封面图片选自 *Wood’s Illustrated Natural History* 一书。

O'REILLY®

可伸缩架构 面向增长应用的高可用

每一天，许多公司都面临着如何去提升关键应用程序规模的问题。随着流量和数据需求的增加，这些应用程序变得越来越复杂和脆弱，从而导致风险上升、可用性降低。本书是一本实践指南，让IT、DevOps和系统稳定性管理员都能了解到，如何避免应用程序在发展过程中变得缓慢、数据不一致或者彻底不可用等问题。

规模增长并不只意味着处理更多的用户，还包括管理更多的风险和保证系统的可用性。作者Lee Atchison在本书中提出了一些基本技巧，使得我们在构建各类应用程序的过程中，既能够保证产品的质量，又能够处理海量的流量、数据以及需求。

本书通过5个部分，分别介绍了以下内容。

- 可用性：你将学习到如何创建高可用的应用程序，以及不断跟踪和提高可用性的技巧。
- 风险管理：你将学习到如何确认、降低和管理应用程序中的风险，测试你的恢复、灾备方案，以及如何构建风险更低的系统。
- 服务和微服务：你将理解服务对大规模运行复杂应用的系统所带来的价值。
- 扩展应用程序：你将学习到如何将服务分配给不同的团队，标识每个服务的关键程度，以及设计故障场景和恢复计划。
- 云服务：理解基于云服务的架构、资源分配以及服务分布。

Lee Atchison是New Relic的首席云架构师和布道师，负责领导搭建公司的基础设施产品，并且帮助公司设计了一个健壮的、基于服务的系统架构。他在Amazon担任高级技术经理的7年中，学习了如何设计基于云的、可扩展的系统，主导创建了AWS Elastic Beanstalk产品。Lee拥有28年的行业经验。

"不要拿你的生意做赌注。规模化的发展是一个不可避免的趋势。本书会告诉你如何切实可行地做到这一点。"

——Colin Bodell
时代公司CTO；Amazon网站应用平台副总裁（2006—2013）

"本书会告诉你，如何在应用程序（以及公司）不断扩张以满足用户日益增长需求的同时，保证一切正常运转。"

——Lew Cirne
New Relic公司CEO

"时刻考虑可能出现的故障情况，是构建大规模应用程序的一个关键因素。本书将帮助你学习如何做到这一点，以及如何在用户增长和公司发展过程中，依然保持应用程序正常运行的一些技巧。"

——Patrick Franklin
Google工程副总裁

SYSTEM ADMINISTRATION

图书分类：系统管理

策划编辑：张春雨
责任编辑：刘舫

Broadview®
www.broadview.com.cn

O'Reilly Media, Inc.授权电子工业出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale in the mainland of China (excluding Hong Kong, Macao SAR and Taiwan)

ISBN 978-7-121-31684-5

定价：65.00元